CodeWarrior®

Profiler Manual

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: 98/08/14 raw

Metrowerks CodeWarrior copyright ©1993–1998 by Metrowerks Inc. and its licensors. All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 9801 Metric, Suite 100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	goto `Metrowerks`

Table of Contents

1 Introduction **7**

Read the Release Notes! 7
Metrowerks Year 2000 Compliance 8
What's New in This Release 8
CodeWarrior and Its Documentation 8
What's in This Manual 9
Where to Go from Here 10

2 Getting Started **13**

System Requirements 13
Installing Profiler 14
Background on Profiling Code 15
What Is a Profiler? 15
Types of Profilers 16
A Profiling Strategy 18
Profiling Code 19

3 Using Profiler **21**

What It Does 21
How It Works 22
Profiling Made Easy 23

4 Viewing Results **31**

What It Does 31
How It Works 31
MW Profiler Window 32
Window Views (Mac OS) 37
Finding Performance Problems 40

5 Profiling Mac OS Code **43**

Profiler Libraries and Interface Files (Mac OS) 43
Profiling Special Cases (Mac OS) 46
Profiling Code with #pragma Statements 47
Profiling Code with Compiler Directives 47
Initializing Profiler with ProfilerInit() 48

Terminating Profiler with ProfilerDump() 51
Profiling Threads 52
Viewing Threads in MW Profiler 53
Profiling Shared Libraries 54
Profilng Code Resources 55
Profiling C++ Static Constructors 55
Profiling MacApp 56
Profilng Asynchronous Routines 57
Profiling Abnormally Terminated Functions 57
Debugging Profiled Code 58
(Mac OS) Profiler Limitations 60
(Mac OS) Profiler Tutorial 60
What You Need 61
Getting Started 61
Moving Ahead 73
Now to Optimize 78
One Last Time 83
Congratulations! 90
Where To Go From Here 91

6 Profiling PlayStation Code 93
Profiler Libraries and Interface Files (PlayStation) 93
Implementing the Profiler Library (PlayStation) 93
Using the Profiler (PlayStation) 97
Total Time and the % columns 98

7 Troubleshooting 99
Profile Times Vary Between Runs 99
Problems while Profiling Inline Functions 100
Profiling Library Could not be Found 101
Profiler and Virtual Memory (Mac OS) 102
Names are Garbled when Viewing a 68K Profile Dump (Mac OS) 102

8 Profiler Reference 105
Compiler Directives (Mac OS) 105

Testing for the Profiler 106
Memory Usage (Mac OS) 107
Time and Timebases (Mac OS) 107
Profiler Menu Reference 109
About MW Profiler 109
File Menu 109
Edit Menu (Mac OS) 111
View Menu 111
Windows Menu (Mac OS) 113
Profiler Function Reference 113
Profiler API 113

Index **127**

Table of Contents

1 Introduction

Welcome to the CodeWarrior Profiler manual. Profiler is a code analysis tool designed to help you make your code more efficient.

The introduction includes the following sections:

- Read the Release Notes!—where to go for critical, last-second details
- Metrowerks Year 2000 Compliance—what you need to know about this software's Year 2000 compliance
- What's New in This Release—new features since the previous release
- CodeWarrior and Its Documentation—a general description of the CodeWarrior architecture and documentation
- What's in This Manual—a description of the contents of this manual
- Where to Go from Here—recommendations for further reading

Read the Release Notes!

Before you use CodeWarrior Profiler, you should read the release notes. They contain important last-minute information about new features, bug fixes, and incompatibilities that *may not be included in the documentation*.

The release notes directory is always included as part of a standard CodeWarrior installation. The release notes directory is also located at the top level of the CodeWarrior CD.

Metrowerks Year 2000 Compliance

The Products provided by Metrowerks under the License agreement process dates only to the extent that the Products use date data provided by the host or target operating system for date representations used in internal processes, such as file modifications. Any Year 2000 Compliance issues resulting from the operation of the Products are therefore necessarily subject to the Year 2000 Compliance of the relevant host or target operating system. Metrowerks directs you to the relevant statements of Microsoft Corporation, Sun Microsystems, Inc., Apple Computer, Inc., and other host or target operating systems relating to the Year 2000 Compliance of their operating systems. Except as expressly described above, the Products, in themselves, do not process date data and therefore do not implicate Year 2000 Compliance issues.

For additional information, visit: `http://www.metrowerks.com/about/y2k.html`.

What's New in This Release

We've improved the stability of the Profiler. This latest version is less prone to crashes.

CodeWarrior and Its Documentation

CodeWarrior is a multi-host, multi-language, multi-target development environment. What does that mean?

Multiple hosts CodeWarrior runs on several different operating systems, including Windows and Mac OS. The features, human interface, and operation of CodeWarrior are very similar on all hosts.

Multiple languages You can use CodeWarrior to program in several languages, including C/C++, Pascal, and Java. Third-party compilers provide support for other languages, such as Fortran. Which languages are available to you depend upon the target for which you are developing software.

Multiple targets You can use CodeWarrior to write software for several different chips or operating systems. CodeWarrior products support programming for a variety of embedded processors and real-time operating systems, the Java Virtual Machine, and desktop operating systems such as Windows, Mac OS, and BeOS.

Most features of CodeWarrior apply regardless of your preferred host, language, or target. General features of CodeWarrior are described in manuals such as the *IDE User Guide* and *Debugger User Guide*.

For a complete understanding of CodeWarrior, you must refer to both the general documentation and the documentation that is specific to your particular target.

While the profiling process is general, each target has its own unique features. This manual describes both the general profiling process and the unique features involved in profiling code for specific targets.

What's in This Manual

The CodeWarrior Profiler is actually a system consisting of three components: the profiler libraries, the calls you add to your code, and the MW Profiler application you use to view results. The Profiler system works with C, C++, and Pascal code.

This manual often refers to the CodeWarrior profiler system as "CodeWarrior profiler," or simply as "the profiler." This manual will specifically refer to the MW Profiler application as "MW Profiler."

Table 1.1 lists every chapter in this manual, and describes the information contained in each. See "CodeWarrior and Its Documentation" on page 8 for a discussion of how these chapters relate to other CodeWarrior documentation.

Table 1.1 Contents of chapters

Chapter	Description
Introduction	this chapter
Getting Started	system requirements, installation instructions, and background information
Using Profiler	a "how to" guide to profiling your code
Viewing Results	a user's guide to displaying profiling results with MW Profiler
Profiling Mac OS Code	information about profiling Mac OS code, including a tutorial
Profiling PlayStation Code	information about profiling PlayStation code
Troubleshooting	common problems and their answers
Profiler Reference	a reference to the libraries, compiler directives, and function calls you use to control the CodeWarrior profiler at compile time

Where to Go from Here

All the manuals mentioned here are available as part of the CodeWarrior documentation included with your product.

If you're new to CodeWarrior Profiler:

- If you are unfamiliar with what a profiler is, read "What Is a Profiler?" on page 15. This section discusses different approaches to profiling code, and why profiling is so useful.
- Read "Using Profiler" on page 21 and "Viewing Results" on page 31. These chapters introduce you to the three components of the CodeWarrior profiler system: the profiler librar-

ies, the calls you add to your code, and the MW Profiler application you use to view results.

If you're experienced with CodeWarrior Profiler:

- For details about profiling Mac OS code, read "Profiling Mac OS Code" on page 43.
- For information about profiling PlayStation code, read "Profiling PlayStation Code" on page 93.
- If you are experiencing problems using any part of the CodeWarrior profiler system, read "Troubleshooting" on page 99.
- For technical details on the CodeWarrior profiler system, including the libraries, API, and other issues, see "Profiler Reference" on page 105.
- For reference on menu items in the MW Profiler application, use "Profiler Menu Reference" on page 109 as your starting point.

For everyone:

- For general information about the CodeWarrior IDE and debugger, see the *IDE User Guide* and the *Debugger User Guide*.
- For information specific to the C/C++ front-end compiler, see the *C Compilers Reference.*
- For information on Metrowerks' standard C/C++ libraries, see the *MSL C Reference* and the *MSL C++ Reference.*
- For information specific to the Pascal front-end compiler, see the *Pascal Compilers Reference.*
- For information on CodeWarrior's standard Pascal library, see the *Pascal Library Reference.*

For general information on profiling Mac OS code:

- Read "Optimizing PowerPC Programs," *Apple Directions*, Nov. 95, p. 17.
- Read "Balance of Power," *develop*, Issues 18 (p. 55) and 19 (p. 17)
- Read "PowerPC Compatibility and Performance Issues," *Tech Note PT38*.

- Read "Taking Extreme Advantage of PowerPC," *MacTech Magazine*, Eric Traut, April 95
- Read "Preparing Your Code for Future PowerPC Processors," *Apple Directions*, May 95, p. 14.

There is a complete index at the back of this manual.

2 Getting Started

This chapter gives you the information you need to get CodeWarrior Profiler installed and running. For new users, this chapter provides a brief overview of profiling code.

This chapter includes the following topics:

- System Requirements — hardware and software requirements
- Installing Profiler— how to install Profiler on your system
- Background on Profiling Code— a brief description of profilers and how to implement them in your code

System Requirements

This section describes what kind of system you'll need in order to use CodeWarrior Profiler.

Profiler is part of a complete package that also includes the CodeWarrior IDE, MW Profiler, and the profiler libraries. You need all these elements to use the Profiler system effectively.

Windows Requirements

MW Profiler for Windows analyzes PlayStation code only.

- An IBM-compatible 486 or higher computer. For best performance, a Pentium-class processor is recommended.
- Windows 95, or Windows NT 4.0 operating system
- A minimum of 16 MB RAM to use the CodeWarrior IDE and debugger along with MW Profiler. 24 MB or 32 MB RAM is preferred. 16 MB may not provide the best performance.
- A CD-ROM drive to install CodeWarrior software, documentation, and examples.

Mac OS Requirements

MW Profiler for Mac OS analyzes Mac OS and PlayStation code.

- A Macintosh-compatible computer with either a 68K or a PowerPC processor. A PowerPC processor is recommended.
- Mac OS System 7.1 or later (for 68K processors) or Mac OS System 7.1.2 or later (for PowerPC processors)
- A minimum of 16 MB RAM to use the CodeWarrior IDE and debugger along with MW Profiler. 24 MB or 32 MB RAM is preferred. 16 MB may not provide the best performance.
- A CD-ROM drive to install CodeWarrior software, documentation, and examples
- For profiling PlayStation code, either the CodeWarrior for Net Yaroze CD or the CodeWarrior for PlayStation CD

Installing Profiler

Your first step toward developing software for your target is to install the CodeWarrior tools. The tools include a variety of components such as the IDE, debugger, plug-in compilers and linkers, standard libraries, runtime libraries and headers, and all necessary documentation.

The CodeWarrior Installer automatically installs all necessary components. It is *strongly* recommended that you use the CodeWarrior Installer to ensure that you have all the required files. If you have any questions regarding the installer, read the instructions built into the CodeWarrior Installer itself.

If you are developing software for the PlayStation, you need to make sure that you have set up your hardware correctly. The *Targeting PlayStation OS* manual describes the extensive hardware setup that is required for CodeWarrior to work properly.

To start the installation process, perform the following steps:

For Windows: On the Windows desktop, double-click the My Computer icon, then double-click the drive that holds the CodeWarrior CD. When you do, a dialog box should appear with a button ti-

tled "Launch CodeWarrior Setup." Click this button. If for some reason you don't see this dialog, double-click the `setup.exe` file located at the root level of the CD.

For Macintosh: On the Macintosh desktop, double-click the icon for the CodeWarrior CD. Then double-click the icon for the CodeWarrior Installer, located at the root level of the CD.

Background on Profiling Code

This section provides you with general information about what a profiler is, different kinds of profilers, and a typical strategy you would follow to measure program performance. Along the way you'll see how the CodeWarrior profiler handles the advantages and disadvantages of profiling code.

Topics discussed are:

- What Is a Profiler?—a brief description of profilers and what they do
- Types of Profilers—different kinds of profilers, their strengths and weaknesses
- A Profiling Strategy—an outline you should follow when profiling your own code
- Profiling Code—three steps to follow when profiling code

What Is a Profiler?

Speed and performance are important issues in most software projects. In most cases, if your code doesn't work quickly, it doesn't work well.

Programmers have regularly observed that 10% of their code does 90% of the work. Reworking code to make it more efficient is a non-trivial task. You should concentrate on improving that core 10% of your code first, and improve the infrequently-used code later, if at all.

Wouldn't it be really cool if you could determine precisely where your code spent its time? That's what a profiler does for you—it

gives you clues. More than clues, the CodeWarrior profiler gives you hard and reliable data.

A good profiler analyzes the amount of time your code spends performing various tasks. Armed with this information, you can apply your efforts to improving the efficiency of core routines.

A profiler can also help you detect bottlenecks—routines your data passes through to get to other places—and routines that are just inordinately slow. Identifying these problems is the first step to solving them.

Types of Profilers

The simplest profilers count how many times a routine is called. They do not report any information about which routines are called by other routines, or the amount of time spent inside the various routines being profiled.

Clearly a good profile of the runtime performance of code requires more information than a raw count. More advanced profilers perform statistical sampling of the runtime environment. These profilers are called passive or sampling profilers.

A passive profiler divides the program being profiled into evenly-sized "buckets" in memory. It then samples the processor's program counter at regular intervals to determine which bucket the counter is in.

The main advantage of a passive profiler is that it requires no modification to the program under observation. You just run the profiler and tell it what program to observe. Also, passive profilers distribute the overhead that they incur evenly over time, allowing the post-processing steps to ignore it. On the other hand, they cannot sample too frequently or the sampling interrupt will overwhelm the program being sampled.

Passive profilers have a significant disadvantage. Although useful, bucket boundaries do not line up with routine boundaries in the program. This makes it difficult if not impossible to determine which *routines* are heavily used. As a result, passive profilers gener-

ate a relatively low-resolution image of what's happening in the program while it runs.

In addition, because they rely on a statistical sampling technique, the program must run for a long enough period to collect a valid sample. As a result, they do not have good repeatability—that is, the results you get from different runs may vary unless the sampling period is long.

The most advanced and accurate profilers are called active profilers. The CodeWarrior profiler is an active profiler.

An active profiler tracks the precise amount of time a program spends in each individual routine, measured directly from the system clock.

To perform this magic, an active profiler requires that you modify the code of the program to be observed. An active profiler gains control at every routine entry and exit. There must be a call to the profiler at the beginning of each profiled routine. The profiler can then track how much time is spent in the routine.

This approach has significant advantages over a passive profiler. An active profiler can report high-resolution results about exactly what your program is doing. An active profiler also tracks the dynamic call tree of a program. This information can be very useful for determining the true cost of calling a routine. The true cost of a routine call is not only the time spent in the routine, it is also the time spent in its children—the subsidiary routines it calls, the routines they call, and so on to whatever depth is necessary.

Because it uses measurements and not statistical sampling, an active profiler is much more accurate and repeatable than a passive profiler.

The requirement that you must modify the actual source code might seem like a significant disadvantage. With the CodeWarrior profiler, this disadvantage is minimal. Activating the profiler for an entire program—or for a range of routines within a program—is simple. The compiler does most of the work, inserting the necessary

calls to the profiler itself. You do have to recompile the project when you turn on profiling.

Active profilers distribute their overhead by routine call frequency. This makes it harder to compensate for the overhead during post processing of the output. The CodeWarrior profiler tracks the overhead automatically, compensates for it, and also reports the overhead to you. You don't have to worry about overhead at all.

Finally, active profilers generate a large amount of raw information. This can lead to confusion and difficulty interpreting the results. The MW Profiler application that is part of the CodeWarrior profiler system handles these difficulties with aplomb. You can view and sort the data in whatever way best suits your needs.

A Profiling Strategy

You use a profiler to measure the runtime performance of your code. What is usually important is how your code's performance measures up to some standard. When approaching the problem of measuring performance, you might want to take these three steps:

1. **Establish your standards.**

   For example, you might decide that you want the program to load in less than ten seconds, or check the spelling of a five-page document that contains no misspellings in 15 seconds. Also decide on the platform you will use for testing, since processor speeds vary.

2. **Determine how to measure time.**

   Your measurement device may be no more complicated than a stopwatch, or you may need to add some simple code to count ticks. At this phase you want to test the code in as close to its finished form as possible, so measure time in a way that is accurate enough to suit your needs, and that has the lowest impact on your code's natural performance. You do not want to run a full-blown profile here, because profiling can add significant overhead, thus slowing down your code's raw performance.

3. **Run the tests and measure results.**

   If you meet your performance goals, your job is done. If your code does not meet your goals, then it's time to profile your code.

Profiling Code

To profile your code, you do three things:

1. **Run a profiler on the area of the code you want tested.**

   This might be a single routine, a group of routines that perform a task, or even the entire application. What you profile depends upon what you are testing.

2. **Analyze the data collected by the profiler and improve your code.**

   You study the results of your profiling and look for problems and room for improvement.

   The profiling process is iterative. You repeat these two steps until you achieve the performance gain you need to meet your goals.

   The rest of this manual discusses how to perform these two steps—profile your code and analyze the results—using the CodeWarrior profiler system.

3. **Retest your code to verify results**

   When you are satisfied that you have reached your goals, you have one more step to perform. You should run your original tests—without the profiler of course—to verify that your code in its natural state meets your performance goals.

   The CodeWarrior profiler will help you meet those goals quickly and easily.

3

Using Profiler

The CodeWarrior profiler lets you analyze how processor time is distributed during your program's execution. With this information, you can determine where to concentrate your efforts to optimize your code most effectively.

This chapter discusses the following principal topics:

- What It Does—an overview of the principle features of the profiler
- How It Works—basic information on the elements of the profiler and about how to use the profiler in your own code
- Profiling Made Easy—a step-by-step guide to using the profiler

What It Does

The CodeWarrior profiler is a state-of-the-art, user-friendly, analytical tool that can profile Pascal, C, or C++ code.

For every project, from the simplest to the most complex, the profiler offers many useful features that help you analyze your code. You can:

- turn the profiler on and off at compile time
- profile any routine, group of routines, or an entire project
- track time spent in any routine
- track time spent in a routine and the routines it calls—its children
- track execution paths and times in a dynamic call tree
- collect detailed or summary data in a profile
- use precision time resolutions for accurate profiling
- (Mac OS) track the stack space used by each routine

How It Works

The CodeWarrior profiler is an active profiler. The profiling system consists of three main components:

- a statically-linked code library of compiled code containing the profiler
- an Application Programming Interface (API) to control the profiler
- the MW Profiler application to view and analyze the profile results

Details of the API are discussed in "Profiler API" on page 113. The MW Profiler application is discussed in "Viewing Results" on page 31.

The rest of this chapter will discuss the general profiling process. Subsequent chapters describe how to carry out the profiling process for your particular target.

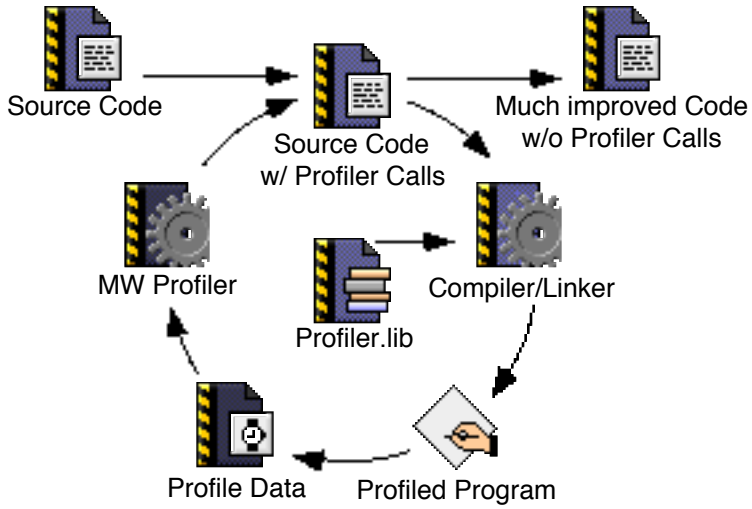
To use the profiler, you do these things:

- Include the correct profiler library and files in your CodeWarrior project
- Modify your source code to make use of the profiler API
- Use the API to initialize the profiler, to dump the results into a file, and to exit the profiler
- Use the MW Profiler application to view the results

You can profile an entire program if you wish or, adding compiler directives to your code, you can profile any individual section of your program.

Figure 3.1 illustrates the flow of events that occurs when you profile code.

Figure 3.1 The profiling process



You modify the original source code slightly to initialize the profiler, dump results, and exit the profiler when through. You may modify the source code more extensively if you wish to profile individual portions of your code.

Then the compiler and linker—using a profiler library—generate a new version of your program, ready for profiling. While it runs, the profiler generates data. Your program will run a little more slowly because of the profiler overhead (sometimes a *lot* more slowly), but that's taken into account in the final results. When complete, you use the MW Profiler application to analyze the data and determine what changes are appropriate to improve performance. You can repeat the process as often as desired until you have turned your code into a fast, efficient, well-oiled machine.

See also "Profiler API" on page 113 and "Viewing Results" on page 31.

Profiling Made Easy

This section takes you step by step through the general process of profiling an application.

To profile an application, you do the following:

1. Add a profiler library to the project
2. Turn on profiling
3. Include the profiler API interface
4. Initialize the profiler
5. Dump the profile results
6. Exit the profiler

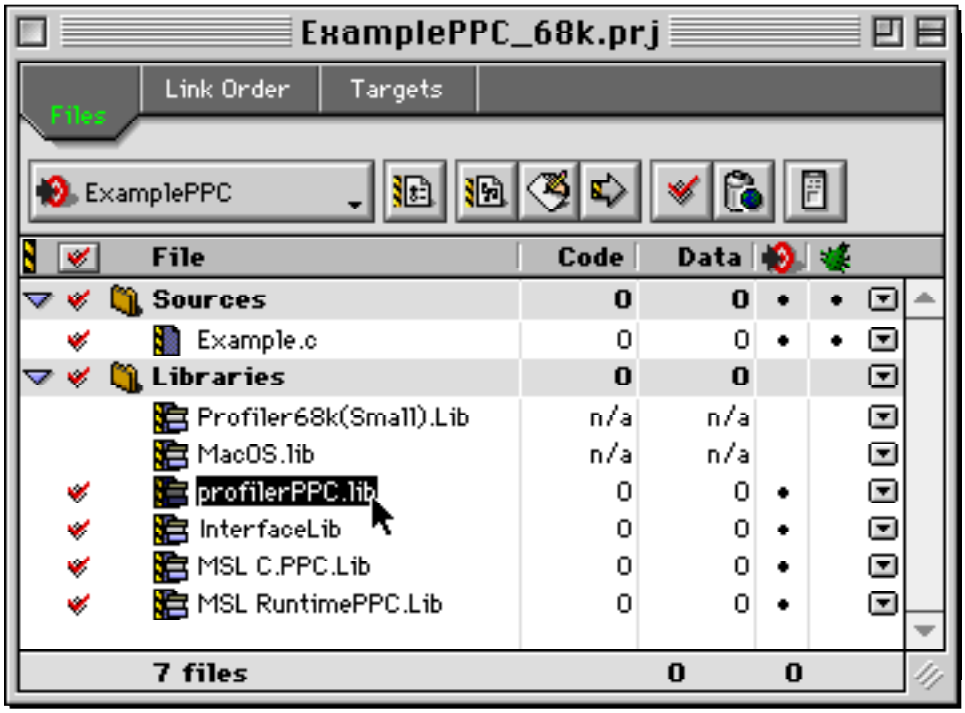
In the steps that follow, we'll detail precisely what to do in both C/C++ and Pascal. These steps may seem a little complicated. Don't be alarmed. Using the CodeWarrior profiler is actually easier than reading about how to do it.

1. Add a profiler library to the project

The code that performs the profiler magic has been compiled into libraries. The precise library that you add to your code depends on the target for which you're profiling code and on the kind of code you're developing. For more information about adding the appropriate libraries for your particular target, read "Profiling Mac OS Code" on page 43 or "Profiling PlayStation Code" on page 93.

For example, Figure 3.2 shows how to add the appropriate library for your code model when profiling code for a Mac OS PowerPC target.

Figure 3.2 The PowerPC profiler library in a C/C++ project



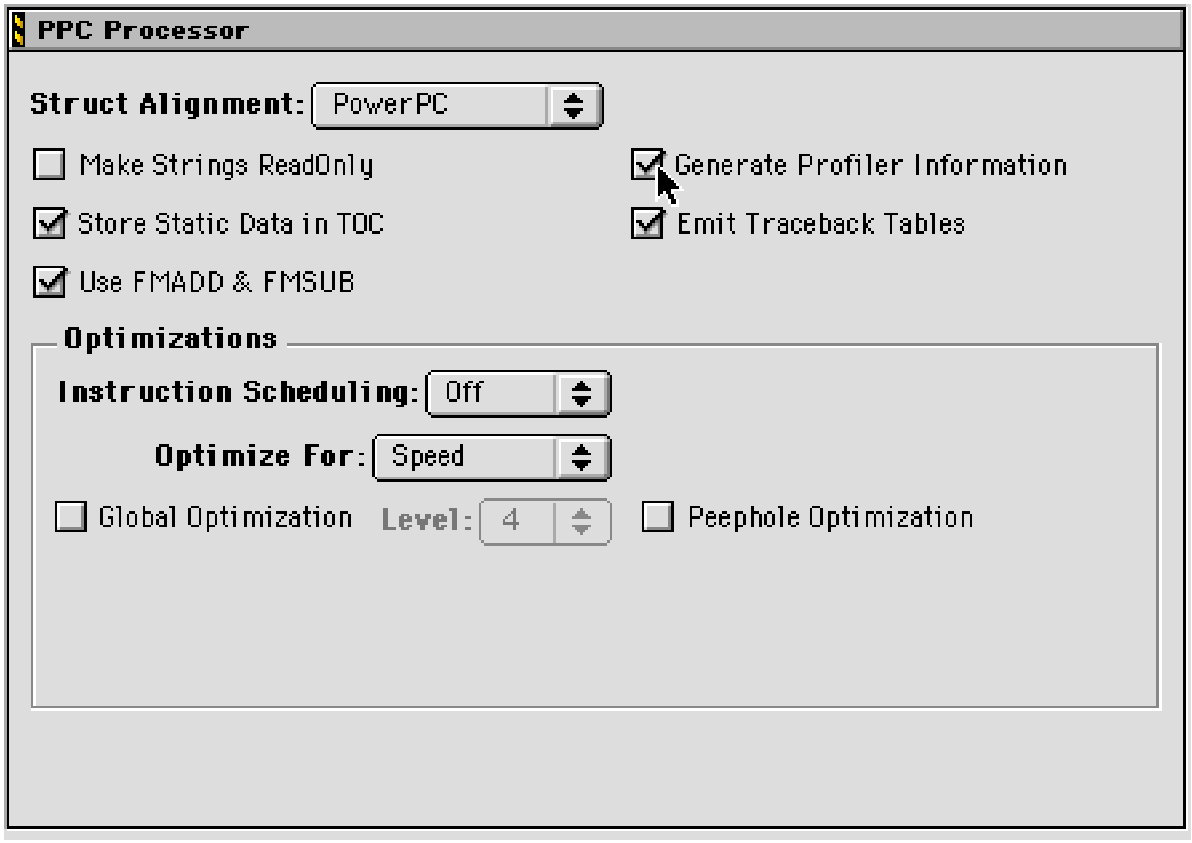
2. Turn on profiling

You can use the following methods to turn profiling on or off:

Project-level profiling (Mac OS)

To turn on profiling for an entire project, use the project settings. In the Project Settings dialog, choose the processor you are generating code for (**68K Processor** or **PPC Processor**) under the Code Generation option. Click the **Generate Profiler Information** checkbox as shown in Figure 3.3. With profiling on, the compiler generates all the code necessary so that every routine calls the profiler.

Figure 3.3 Processor Preferences options for PowerPC



Routine-level profiling

To profile certain routines (rather than the entire project), use the appropriate profiler API calls for your target to initialize the profiler, set up profiling, and immediately turn profiling off. You can then manually turn profiling on and off by placing profiler calls around the routine or routines you want to profile. For example, you could modify Mac OS code to look like Listing 3.1.

Listing 3.1 profiling a routine

```
void main()
{
  ...
  err = ProfilerInit(...);
```

```
  if (err == noErr)
  {
    ProfilerSetStatus(FALSE); // turn off profiling until needed.
    // more code....

    // now you reach routine you want to profile
    ProfilerSetStatus(TRUE);  // turn on profiling
    foobar();  // this routine is profiled and shows up in viewer
    ProfilerSetStatus(FALSE);  // turn profiling off again

    // more code....
    ProfilerTerm();
  }
}
```

Assuming that profiling is on for an entire project, you can turn off profiling at any time. First, use an appropriate call to turn off profiling. Then use another call to turn it on. Turn it on just before calling the routine or routines you are interested in. Turn it off when those routines return. It's really that easy.

Alternatively, you can use #pragma statements in C/C++, or Pascal compiler directives. These aren't as useful as using profiler API calls. For example, suppose you have two routines—`foo()` and `bar()`—that each call a third utility routine, `barsoom()`. If you use compiler directives to turn on profiling for `foo()` and `barsoom()`, the result you get will include the time for `barsoom()` when called from `bar()` as well.

3. **Include the profiler API interface**

   To use the profiler, you add at least three profiler-related calls to your code. These calls are detailed in the next three steps. The process varies slightly for the different languages and targets.

   ### Including the profiler API for C/C++

   Source files that make calls to the profiler API must include the appropriate header file for your target. For example, to profile an entire Mac OS application, you would add this line of code to the source file that includes your `main()` function:

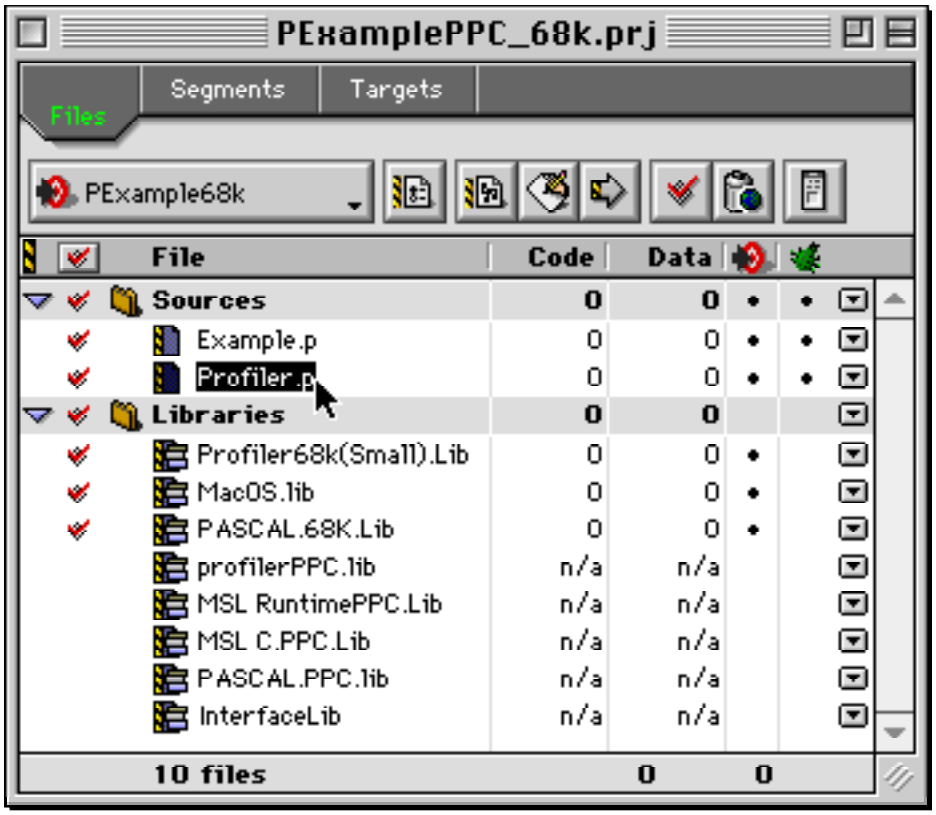
```
#include <profiler.h>
```

TIP: You don't have to include the header file in every file that contains a profiled function, only in those that actually make direct profiler API calls.

Including the profiler API for Pascal

Add the Profiler.p file to your project, as shown in Figure 3.4.

Figure 3.4 The Profiler.p file in a Pascal project



4. **Initialize the profiler**

At the beginning of your code, you call the appropriate function for your target. See "Profiler API" on page 113 to find out the precise function name that you'll need for your specific target.

NOTE: If you are developing code for the PlayStation, you must first create a buffer to store profile data before you can take advantage of the profiler. See "Create a buffer in your application" on page 94 to find out how to create this buffer for the profiler.

5. **Dump the profile results**

Obviously, if you profile code you want to see the results. The profiler dumps the results to a data file. The data is in a proprietary format understood by the MW Profiler viewing application.

TIP: Although the original profiler data file is in a proprietary format, you can use the **Save As...** command in the MW Profiler **File** menu to save the data in a tab-delimited text file.

6. **Exit the profiler**

When you are all through with the profiler, before exiting the program you *must* terminate the profiler by calling the correct profiler API function. If you initialize the profiler and then exit the program without terminating the profiler, timers may be left running that could crash the machine.

The call to terminate the profiler stops the profiler and deallocates memory. It does not dump any information. Any collected data that has not been dumped is lost when you call the function to terminate the profiler.

Having performed these quick steps, you simply compile your program and run it. When you quit, the results will be waiting for your analysis. MW Profiler, the viewing application, is introduced in "Viewing Results" on page 31.

In summary, the process of using the CodeWarrior profiler is quite easy. You add the requisite library, turn on profiling, include the header file, initialize the profiler, dump the results, and exit. It is a remarkably painless and simple process that quickly gets you all the data you need to perform a professional-level analysis of your application's runtime behavior.

4 Viewing Results

This chapter discusses MW Profiler—the application you use to view the data created by the CodeWarrior profiler.

In this chapter you will look at:

- What It Does—the principle features of MW Profiler
- How It Works—the MW Profiler interface and how you can view data
- Finding Performance Problems—use the MW Profiler to locate problems

What It Does

The MW Profiler viewer application displays profiler output for you to analyze the results of your program's execution. MW Profiler reads the profiler dump files created by the calls in your code and displays the data in a form that you can use. Using the data display you can:

- sort data by any of several relevant criteria such as name, time in routine, percent of time in routine, and so forth
- open multiple profiles simultaneously to compare different versions of the profiled code
- identify trouble spots in the code
- (Mac OS) view summary, detailed, or object-based data

The MW Profiler application is very easy to use.

How It Works

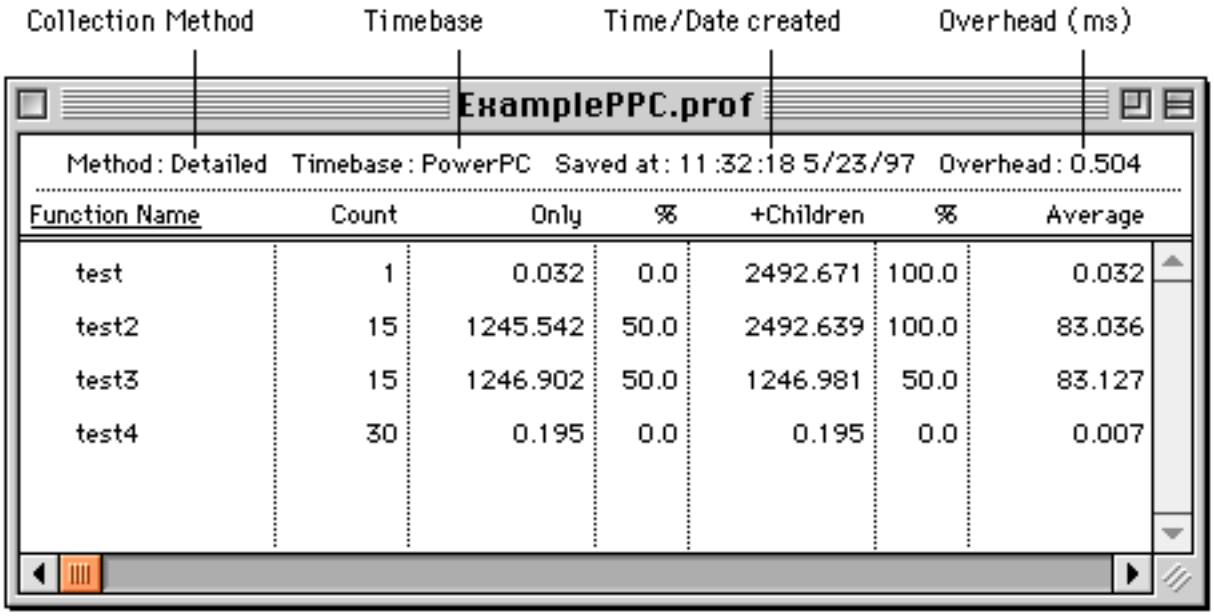
You open profile data files exactly as you open files in any application. You can use the **Open...** command from the MW Profiler **File**

menu or drop the data file's icon on the MW Profiler icon. Whatever approach you take, when you open a file a window appears.

MW Profiler Window

MW Profiler allows you to view several elements of the profile data simultaneously, as shown in Figure 4.1.

Figure 4.1 Profiler window


Collection Method
Timebase
Time/Date created
Overhead (ms)
ExamplePPC.prof
Method: Detailed Timebase: PowerPC Saved at: 11:32:18 5/23/97 Overhead: 0.504
Function Name Count Only % +Children % Average
test 1 0.032 0.0 2492.671 100.0 0.032
test2 15 1245.542 50.0 2492.639 100.0 83.036
test3 15 1246.902 50.0 1246.981 50.0 83.127
test4 30 0.195 0.0 0.195 0.0 0.007

Profiler window header items (Mac OS)

MW Profiler allows you to view summary data, detailed data, or object-based data. All three ways of viewing data use the same window shown in Figure 4.1. The display of the contents changes slightly for the different views.

See also "Window Views (Mac OS)" on page 37.

At the top of the window is a line with information about the data on display. The figure shows four items, but six are possible. What they are and what they represent is listed in Table 4.1.

Table 4.1 Profile window header items

Item	Contents
Method	the collection method, detailed or summary
Timebase	the time base that was used to collect the profile information
Saved at	the date and time the profile data was created
Overhead	time (in milliseconds) used by the profiler
OverflowStack	the number of routines that were called when the profiler's internal stack was overflowing
OverflowFunctions	the number of routines called that did not fit in the profiler's routine table

The last two items—OverflowStack and OverflowFunctions—only appear if the parameters you passed to `ProfilerInit()` did not create a buffer large enough to hold all the collected data.

The OverflowStack number includes duplicate routine calls. If it is small, it is the amount by which you should increase the `stackDepth` parameter in the call to `ProfilerInit()`. If it is large, increase the value of the `stackDepth` parameter by some increment and try again.

The OverflowFunctions number also includes duplicate routine calls. It is likely to be much larger than the size by which you need to increase the `numFunctions` parameter in the call to `ProfilerInit()`. It might even be in the thousands.

If either overflow occurs, just increase the value of the respective parameter in the `ProfilerInit()` call so the profiler allocates a larger data buffer.

See also "ProfilerInit()" on page 114.

Profiler window data columns

Below the header information is a series of columns containing data from the profile. All times are displayed according to the resolution

of the timer that you use to profile data. The results in the window are only as precise as the timer used.

Mac OS The times shown in the data columns are displayed in milliseconds. A millisecond (ms) is one thousandth of a second. Each time datum is reported to three decimal places. This gives you microsecond results. A microsecond (μs) is one thousandth of a millisecond, or one millionth of a second. However, some time bases (most notably `ticksTimeBase`) are less precise. Ticks increment 60 times a second. See "Time and Timebases (Mac OS)" on page 107.

Resizing data columns

Figure 4.1 does not show all the data columns. You can scroll to see the columns beyond the right edge of the window. You can also resize the columns to get more information in the window. The method by which you resize the data columns depends on the host you are using to run MW Profiler:

For Windows: Move the cursor in between two of the data column names. The cursor will change to a directional arrow, similar to the one shown in Figure 4.2. Click and drag to resize the data columns to the size that you want.

For Mac OS: Move the cursor over one of the column lines. The cursor will change to a directional arrow, as shown in Figure 4.2. Click and drag the column line to where you want it.

Figure 4.2 Resize columns

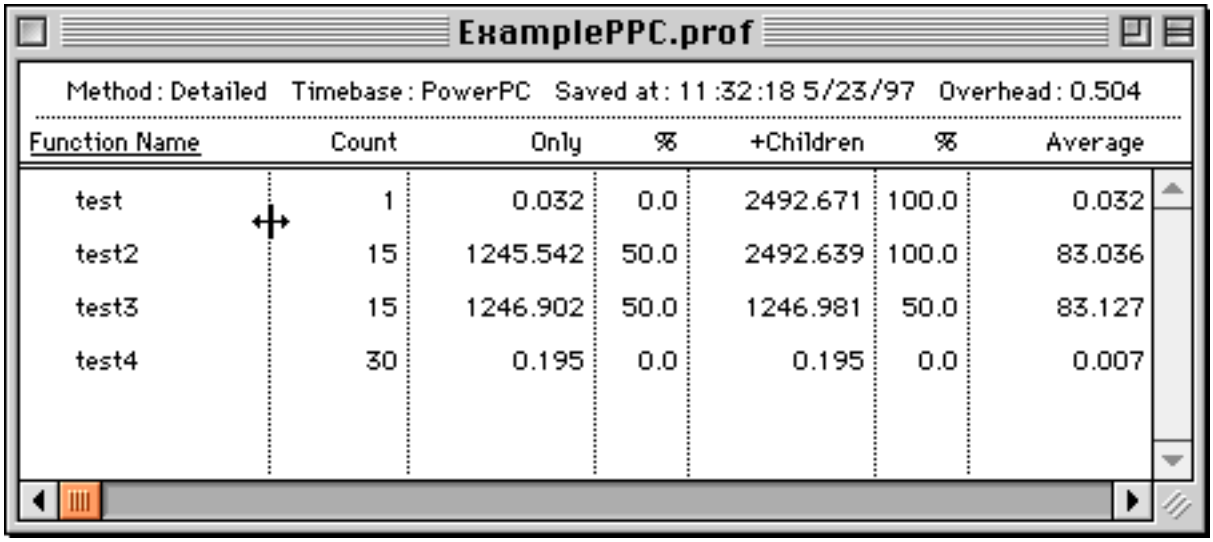


Table 4.2 lists each of the columns in the profiler window (from left to right) and the information that column contains.

Table 4.2 Profile window data columns

Column	Contents
Function name	the routine name (if it is a C++ function name, the profiler unmangles it)
Count	the number of times this routine was called
Only	time spent in the routine itself without counting any time in routines called by this routine
%	percent of total time for the Only column
+Children	time spent in this routine and all the routines it calls.
%	percent of total time for the +Children column
Average	the average time for each routine invocation. (the Only time divided by the number of times the routine was called)

Column	Contents
Maximum	the longest time for an invocation of the routine
Minimum	the shortest time for an invocation of the routine
Stack Space	(Mac OS) the largest size (in bytes) of the stack when the routine is called (for 68K code the actual size may be slightly larger)

NOTE: If you're profiling Mac OS code, you should know that there is a slight difference between the 68K and the PowerPC in the stack tracking code. On a 68K machine the profiler is called before a routine sets up its stack. Therefore, the stack values that a 68K profile reports *do not include the called routine's stack*. On a PowerPC, the profiler is called after a routine sets up its stack. Therefore, the stack values on a PowerPC profile do include the called routine's stack.

Sorting data

You can view the data sorted by the value in any column. Names are listed alphabetically from top to bottom, numbers are listed from greater to lesser.

To change the sort order, you do the following:

For Windows: Simply click the column title. The data is sorted by the value in the column that you clicked.

For Mac OS: Either click the column title, or choose a menu item from the **View** menu. The heading becomes underlined and data is sorted by the value in that column.

Exporting data

If you wish to export information from the results, use the **Save As...** command in the **File** menu. This saves all the data in a tab-delimited text file. The text file is organized the same way that the window is organized.

Multiple windows

You can open any number of different profile windows simultaneously, limited only by the MW Profiler's memory allotment. This allows you to compare the results of different runs easily.

Window Views (Mac OS)

In the MW Profiler **View** menu you may choose to view the data in one of three ways: summary, detailed, or object. Not all possibilities are available for all profiles.

Summary View

The summary view displays a complete, non-hierarchical, flat list of each routine profiled. No matter what calling path was used to reach a routine, the profiler combines all the data for the same routine and displays it on a single line. Figure 4.3 shows a summary view.

Figure 4.3 Summary view

ExamplePPC.prof

Method: Detailed Timebase: PowerPC Saved at: 11:32:18 5/23/97 Overhead: 0.504

Function Name	Count	Only	%	+Children	%	Average
test	1	0.032	0.0	2492.671	100.0	0.032
test2	15	1245.542	50.0	2492.639	100.0	83.036
test3	15	1246.902	50.0	1246.981	50.0	83.127
test4	30	0.195	0.0	0.195	0.0	0.007

The summary view is particularly useful for comparing routines to see which take the longest time to execute. The summary view is also useful for finding a performance problem with a small routine that is called from many different places in the program. This view

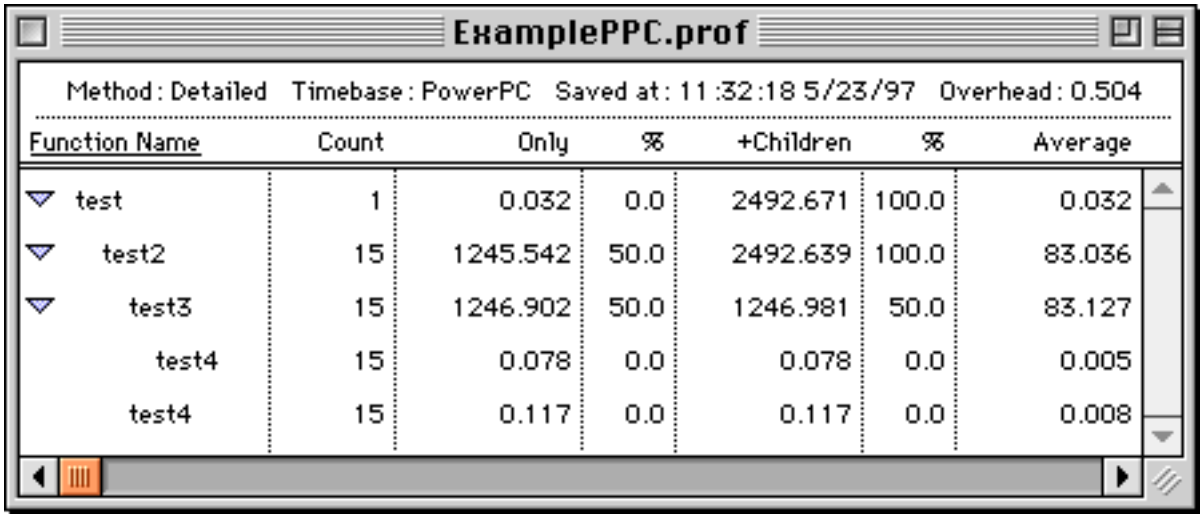
helps you look for the routines that make heavy demands in time or raw number of calls.

A summary view window can be displayed for any profile.

Detailed View

The detailed view displays routines according to the dynamic call tree as shown in Figure 4.4.

Figure 4.4 Detailed view



Routines that are called by a given routine are shown indented under that routine. This means that a routine may appear more than once in the profile if it called from different routines. This makes it difficult to tell how much total time was spent in a routine. However, you can use the summary view for that purpose.

The detailed view is useful for detecting design problems in code. MW Profiler in detailed view lets you see what routines are called how often from what other routines. Armed with knowledge of your code's underlying design, you may discover flow-control problems.

For example, you can use detailed view to discover routines that are called from only one place in your code. You might decide to fold

that routine's code into the caller, thereby eliminating the routine call overhead entirely. If it turns out that the little routine is called thousands of times, you can gain a significant performance boost.

You can use the **Expand All** or **Collapse All** commands in the **View** menu to open or close the entire hierarchy at once.

In detailed view, sorting is limited to routines at the same level in the hierarchy. For example, if you sort by routine name, the routines at the top of the hierarchy will be sorted alphabetically. For each of those first-level routines, its second-level routines will be sorted alphabetically underneath it, and so on.

The detailed view requires that `collectDetailed` be passed to `ProfilerInit()` when collecting the profile. If `collectSummary` is used, you cannot display the data in detailed view.

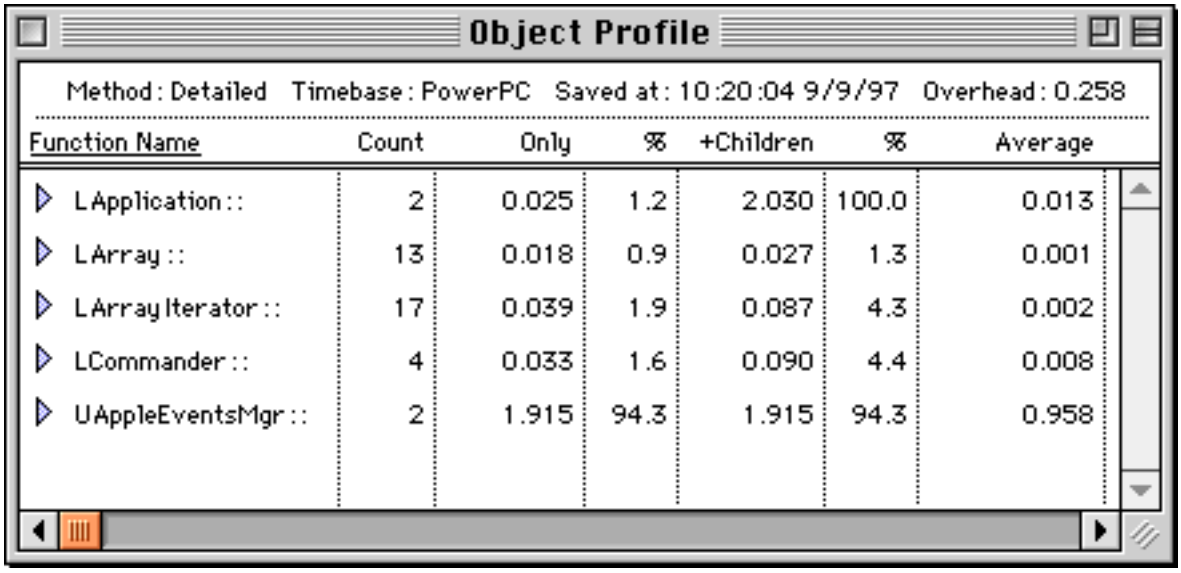
Object View

The object view displays summary information sorted by class. Beneath each class the methods are listed. This is a two-level hierarchy. You can open and close a class to show or hide its methods, just like you can in the detailed view.

When sorting in object view, functions stay with their class, just like subsidiary functions in detailed view stay in their hierarchical position. Figure 4.5 shows the methods sorted by count.

Object view allows you to study the performance impact of substituting one implementation of a class for another. You can run profiles on the two implementations, and view the behavior of the different objects side by side. You can do the same with the summary view on a routine-by-routine basis, but the object view gives you a more natural way of accessing object-based data. It also allows you to gather all the object methods together and view them simultaneously, revealing the effect of interactions between the object's methods.

Figure 4.5 Object view


Object Profile
Method: Detailed Timebase: PowerPC Saved at: 10:20:04 9/9/97 Overhead: 0.258
Function Name Count Only % +Children % Average
LApplication:: 2 0.025 1.2 2.030 100.0 0.013
LArray:: 13 0.018 0.9 0.027 1.3 0.001
LArrayIterator:: 17 0.039 1.9 0.087 4.3 0.002
LCommander:: 4 0.033 1.6 0.090 4.4 0.008
UAppleEventsMgr:: 2 1.915 94.3 1.915 94.3 0.958

Object view will display "N/A" (Not Available) in the +Children column for classes in a `collectSummary` profile. This is because the detail information is missing from the file.

The object view requires that the profile contain at least one mangled C++ name. If there is none, you cannot use object view.

Finding Performance Problems

As you work with MW Profiler, you will see that the information provided quickly guides you to problem areas.

To look for time hogs, sort the view by either the Only column or the +Children column. Then examine routines that appear near the top of the list. These are the routines that swallow the greatest percentage of your code's time. Any improvement in these routines will be greatly magnified in your code's final performance.

You may also want to sort based on the number of times a routine is called. The time you save in a heavily-used routine is saved each time it is called.

Mac OS: If stack size is a concern in your code, you can sort based on the Stack Space column. This lets you see the largest size the stack reached during the profile. Remember that in Mac OS 68K code the stack for the called routine is not included in the Stack Space data.

5 Profiling Mac OS Code

This reference section discusses profiling Mac OS code and associated special cases.

The sections in this chapter are:

- Profiler Libraries and Interface Files (Mac OS)—the libraries and interface files that you add to your code in order to use the profiler
- Profiling Special Cases (Mac OS)—special cases to consider when profiling code
- (Mac OS) Profiler Limitations—important information for profiling 68K code
- (Mac OS) Profiler Tutorial—a hands-on tutorial on how to use the profiler to find problem areas in your code

Profiler Libraries and Interface Files (Mac OS)

You can find all of the profiler libraries and interface files in the Profiler folder in the Metrowerks Utilities folder. The profiling code that actually keeps track of the time spent in a routine exists in a series of libraries. Depending upon the nature of your project and the platform for which you are writing code, you link in one or another of these libraries as appropriate.

See also "Add a profiler library to the project" on page 24 and "Profiling Special Cases (Mac OS)" on page 46.

The profiler libraries require the Macintosh operating system. You must have the appropriate Mac OS library added to your project as listed in Table 5.1. Your project may need other libraries as well.

Table 5.1 Mac OS libraries

Language	68K	PowerPC
C/C++	`MacOS.lib`	`InterfaceLib`
Pascal	`Pascal/MacOS.lib`	`InterfaceLib`

Table 5.2 lists the libraries you use for profiling. Both Pascal and C use the same libraries. For 68K code, use the "Small" library for projects using the near code model, and "Large" for the far code and smart code models.

Table 5.2 Profiler libraries

Library	Processor	When To Use
`profiler68k(Small).lib`	68K (A5 globals)	near/small model applications
`profiler68k(Large).lib`	68K (A5 globals)	far/large/smart model applications
`profiler68kA4(Small).lib`	68K (A4 globals)	near/small model code resources
`profiler68kA4(Large).lib`	68K (A4 globals)	far/large/smart model code resources
`Profiler68kCFM.lib`	68K (A5 globals)	code fragments and shared libraries

Library	Processor	When To Use
`ProfilerLIB`	CFM 68K & PowerPC	code fragments and shared libraries
`profilerPPC.lib`	PowerPC	PowerPC applications and shared libraries

TIP: `ProfilerLIB` is a FAT shared library. Use `ProfilerLIB` when you want to collect information across shared library boundaries. If all you want is data from one library, use one of the static link libraries and link it into your project. For example, say you split your application into an application and 2 shared libraries and want to profile it. You can link a static link library to each part and get three reports, one for each piece. But that data is very hard to interpret because each report is missing some context that happens to be included in another report. By using the shared library version of the profiler, you will get one report that spans all three components.

The profiler also has separate interface files for C/C++ and Pascal. They are listed in Table 5.3.

Table 5.3 Profiler interface files

`profiler.h`	The header file for the profiler API for C and C++. Include this file to make calls to control the profiler.
`profiler.p`	The interface file for the profiler API for Pascal. Include this file to make calls to control the profiler.

Profiling Special Cases (Mac OS)

The profiler handles recursive and mutually recursive calls transparently. The profiler also warns you when profiling information was lost because of insufficient memory. (The profiler uses memory buffers to store profiling data.)

For leading-edge programmers, the profiler transparently handles and reliably reports the times for asynchronous completion routines, multiple threads, 68K code resources, PowerPC shared libraries, and even abnormally terminated routines exited through the C++ exception handling model (try, throw, catch) or the ANSI C library `setjmp()` and `longjmp()` routines.

C++ programmers may wish to consult the Metrowerks PowerPlant documentation for information on the `StProfileSection` class. This class automates the process of initializing the profiler, dumping results, and exiting the profiler. You do not need to use the rest of PowerPlant to take advantage of the `StProfileSection` class.

This section describes the special cases that are specific to profiling Mac OS code:

- Profiling Code with #pragma Statements
- Profiling Code with Compiler Directives
- Initializing Profiler with ProfilerInit()
- Terminating Profiler with ProfilerDump()
- Profiling Threads
- Viewing Threads in MW Profiler
- Profiling Shared Libraries
- Profilng Code Resources
- Profiling C++ Static Constructors
- Profiling MacApp
- Profilng Asynchronous Routines
- Profiling Abnormally Terminated Functions
- Debugging Profiled Code

Profiling Code with `#pragma` Statements

You can substitute `#pragma` statements for profiler API function calls to profile your C/C++ code on the function level. However, this is not as useful as the profiler calls. See "Routine-level profiling" on page 26 for more information.

Setting the "Generate Profiler Calls" Processor preference option sets a preprocessor variable named `__profile__` to 1. If profiling is off, the value is zero. You can use this value at compile time to test whether profiling is on.

Instead of, or in addition to, setting the option in the Processor preferences, you can turn on profiling at compile time. The C/C++ compiler supports three preprocessor directives that you can use to turn compiling on and off at will.

`#pragma profile on`	enables calls to the profiler in functions that are declared following the pragma
`#pragma profile off`	disables calls to the profiler in functions that are declared following the pragma
`#pragma profile reset`	sets the profile setting to the value selected in the preferences panel

You can use these directives to turn profiling on for any functions you want to profile, regardless of the settings in the Processor preferences. You can also turn off profiling for any function you don't want to profile.

Profiling Code with Compiler Directives

You can substitute compiler directives for profiler API function calls to profile your Pascal code on the routine level. However, this is not as useful as the profiler calls. See "Routine-level profiling" on page 26 for more information.

The Pascal compiler implements two directives to control the generation of calls to the profiler at compile time.

`{$PROF +}`	enables calls to the profiler in procedures and routines that are declared following the directive
`{$PROF -}`	disables calls to the profiler in procedures and routines that are declared following the directive

The status of the profiler directive is saved and restored with the `{$PUSH}` and `{$POP}` directives.

Use these directives to turn profiling on for any routines you want to profile, regardless of the settings in the Processor preferences. You can also turn off profiling for any routine you don't want to profile.

Initializing Profiler with `ProfilerInit()`

At the beginning of your code, you call `ProfilerInit()` to initialize the Profiler. Table 5.4 shows the prototypes for `ProfilerInit()` for both C/C++ and Pascal.

Table 5.4 ProfilerInit() prototypes

Language	Prototype

C/C++	`pascal OSErr ProfilerInit(`
	`  ProfilerCollectionMethod method,`
	`  ProfilerTimeBase timeBase,`
	`  short numFunctions, short`
	`  stackDepth);`

Pascal	`FUNCTION ProfilerInit(`
	`  method: ProfilerCollectionMethod;`
	`  timeBase: ProfilerTimeBase;`
	`  numFunctions: INTEGER;`
	`  stackDepth: INTEGER):OSErr;`

The parameters tell the profiler how this collection run is going to operate, and how much memory the profiler should allocate for its data buffers. Each parameter and its purpose is given in Table 5.5.

Table 5.5 ProfilerInit() parameters

Parameter	Purpose
`method`	collect detailed or summary data
`timeBase`	time scale to use in measurements
`numFunctions`	maximum number of routines to profile
`stackDepth`	approximate maximum depth of deepest calling tree

The collection `method` may be either `collectDetailed` or `collectSummary`. If you collect detailed data, you get information for the calling tree—the time in each routine and each of its children in the calling hierarchy. Summary data collects data for the time spent in each routine without regard to the calling chain. Collecting detailed data requires more memory.

The `timeBase` may be—in order of decreasing precision—`bestTimeBase, PPCTimeBase, microsecondsTimeBase, timeMgrTimeBase`, or `ticksTimeBase`. The `bestTimeBase` option automatically selects the most precise timing mechanism available on the computer running the profiled software. `PPCTimeBase` is only available with PowerPC chips.

The `numFunctions` parameter is the approximate number of routines to be profiled. The `stackDepth` parameter is the approximate maximum depth of your calling chain. You don't need to know the precise values ahead of time. If the profiler runs out of memory to hold data in its buffers, it loses some data but you'll be told in the results. You can then modify the parameters in the call to `ProfilerInit()` to increase the buffers and preserve all your data.

The profiler allocates buffers in temporary memory based on the method of collection, the number of routines, and the depth of the calling tree. Using temporary memory reduces the effect that the profiler has on the application's memory partition.

The call to `ProfilerInit()` returns a non-zero error value if the call fails for any reason. Use the return value to ensure that memory

was allocated successfully before continuing with the profiler. Typically you would add this call as conditionally compiled code so that it compiles and runs only if profiling is on and the call to `ProfilerInit()` was successful.

You call `ProfilerInit()` before any profiling occurs. Typically you make the call at the beginning of your code.

See also "Time and Timebases (Mac OS)" on page 107 and "Memory Usage (Mac OS)" on page 107.

Calling `ProfilerInit()` in C/C++

In C/C++, the call would be at the beginning of your `main()` function.

The call might look like this:

```
if (!ProfilerInit(collectDetailed, bestTimeBase, 20, 5))
{
// your profiled code
}
```

Of course, your parameters may vary depending upon how many routines you have and the depth of your calling chains.

See also "Profiling C++ Static Constructors" on page 55.

Calling `ProfilerInit()` in Pascal

Call `ProfilerInit()` at the beginning of your application. In Pascal, it might look like this:

```
IF ProfilerInit(collectDetailed,
    bestTimeBase, 20, 5) = noErr THEN
BEGIN
{ your profiled code }
END
```

Of course, your parameters may vary depending upon how many routines you have and the depth of your calling chains.

Terminating Profiler with `ProfilerDump()`

The profiler dumps its data to a file when you call `ProfilerDump()`. The file appears in the current default directory, usually the project directory.

You provide a file name when you call `ProfilerDump()`. You may dump results as often as you like. You can provide a different file name for intermediate results (if you have multiple calls to `ProfilerDump()`), or use the same name. If the specified file already exists, a new file is created with an incrementing number appended to the file name for each new file. This allows the dump to be called inside a loop with a constant file name. This can be useful for dumping intermediate results on a long task.

`ProfilerDump()` does not clear accumulating results. If you want to clear results you can call `ProfilerClear()`.

A typical call to `ProfilerDump()` would be placed just before you exit your program, or at the end of the code you are profiling. The prototypes for `ProfilerDump()` are listed in Table 5.6.

Table 5.6 ProfilerDump() prototypes

Language	Prototype
C/C++	`OSErr ProfilerDump(` `  unsigned char *filename);`
Pascal	`FUNCTION ProfilerDump(` `  filename : Str255):OSErr;`

Calling `ProfilerDump()` in C/C++

There is only one parameter. It must be a pointer to a Pascal-style string that becomes the file name for your results. A typical call might look like this:

```
error = ProfilerDump("\pMyCode.prof");
```

Calling `ProfilerDump()` in Pascal

There is only one parameter, an `Str255` that becomes the file name for your results. A typical call might look like:

```
error := ProfilerDump('MyCode.prof');
```

WARNING! Calls to `UnloadSeg()` in a 68K application cause serious problems for the profiler. The profiler maintains pointers to locations in the code. If you unload a segment that has profiled routines, that code is free to move around in memory, or be purged. At that time the profiler's internal pointers become invalid. This is the leading cause of corrupted profile data files.

Profiling Threads

You may use the profiler to study cooperative threads. The profiler does not support preemptive threads.

To profile a cooperative thread, you do a little setup work when you create the thread. You call `ProfilerCreateThread()` to tell the profiler to allocate the necessary buffers. (This doesn't actually create a thread, just the necessary structures so the profiler can keep track of a thread.) You provide three parameters. You specify the number of routines, the byte size of the stack created for the thread, and provide the address where the profiler can return a reference to this thread's profile data. You must keep track of this reference for use in other thread-related profiler calls.

When your thread gains control and executes, its SwapIn callback routine is called. In that routine, call `ProfilerSwitchTo-Thread()` to tell the profiler that this thread is now running. You provide the profiler's thread reference you got when you called `ProfilerCreateThread()`.

NOTE: Do not profile the swapIn routine itself. Thread callback procs (swapIn & swapOut for example) should be compiled to not call the profiler.

When your thread finishes, call `ProfilerDeleteThread()`. You provide the profiler's thread reference to identify which thread is involved. This call cleans up the profiler's thread-tracking structures. You do not need to dump your data before calling `ProfilerDeleteThread()`. The data collection buffers are separate from the memory used to track each thread.

You do not need to call `ProfilerCreateThread()` for the implicit main thread created for your process. However, if you profile code in some other thread and then want to profile the main thread, you will need to call `ProfilerSwitchToThread()` to resume profiling the main thread. You can get the necessary thread reference by calling `ProfilerGetMainThreadRef()`.

TIP: If you are programming in C++, you may want to consider using the PowerPlant `LThread` class. This class has built-in support for profiling threads. See the PowerPlant source code and documentation for more details on this class.

Viewing Threads in MW Profiler

If you are profiling a multi-threaded application, you should be aware of how the results are shown in MW Profiler.

In summary mode, the profiler will mix all the threads together, and you can't tell what is called from what thread. Object mode does the same as summary mode.

In detailed mode, if the threads have a different top-level procedure (the procedure you register with the Thread Manger), then the profiler will show them as separate top-level entries. Threads with the same top-level name will be combined together. For example, say you have a program whose `main()` initializes the profiler, starts up 3 threads, and then calls a function to run event loop.

Listing 5.1 Profiling multi-threaded code

```
main()
{
  ProfilerInit(collectDetailed,bestTimeBase,20,5);

  //...Create a DoBackGroundWork thread
  //...Create another DoBackGroudWork thread
  //...Create a thread that runs DoIdleStuff

  DoMainEventLoop();

  ProfilerDump('test");
  ProfilerTerm();
}
```

The profile output file for this will have three top-level functions listed in detail mode: `DoBackGroundWork`, `DoIdleStuff`, and `DoMainEventLoop`. The two threads that are running `DoBackGroundWork` would combine their profile information.

Profiling Shared Libraries

The profiler supports profiling shared libraries for PowerPC and CFM 68K.

Add the shared library named ProfilerLIB to your project instead of the static link library ProfilerPPC.lib. Include the same header or interface (`profiler.h` or `Profiler.p`).

The shared library profiler collects data from an application and any shared libraries simultaneously. This allows you to split an application into multiple shared libraries and a smaller application, but still profile as if all the components were one large application. Each shared library you create must be compiled with **Generate Profiler Calls** selected in the Code Generation pane in the Project Settings dialog.

NOTE: If you are compiling code for the PowerPC, click the **Emit Profiler Calls** checkbox in the Code Generation pane.

See also "Turn on profiling" on page 25.

Profilng Code Resources

The profiler supports profiling code resources.

For 68K code resources, add the Profiler68kA4.lib library to your project instead of Profiler68k.lib. Include the same header or interface (`profiler.h` or `Profiler.p`).

The code resource must not move in memory between the `ProfilerInit()` and `ProfilerTerm()` calls. This is because the A4 globals can move. The profiler has a pointer to them that is not updated except when you call `ProfilerInit()`.

For PowerPC code resources, you may use either the static link library ProfilerPPC.lib or the shared library ProfilerLIB. You must make sure that the PowerPC code resource does not move in memory while profiling. This means that profiling accelerated resources under applications that unlock the code resource after each call is not supported.

Profiling C++ Static Constructors

It is possible to profile static constructor calls if the profiler is initialized before the static constructor is called. If you are working with shared libraries, you need to determine the order in which the shared libraries are loaded. Put the initialize call to the profiler in the first one loaded.

To profile the static constructors called from `__sinit()`, the profiler must be initialized before calling `__sinit()`.

To do this with the 68K compiler, use the function `__PreInit__(void)` declared in Listing 5.2.

Listing 5.2 __myinit: Initializing the 68K compiler in C++

```
extern "C" void __PreInit__(void);
void __PreInit__(void)
{
  OSErr error;
  error=ProfilerInit(collectDetailed,
                  bestTimeBase, 20, 5);
}
```

Of course, you'll set the parameters in the call to `ProfilerInit()` appropriately for your work.

For PowerPC code, write a small routine `__myinit` that initializes the profiler, then calls `__sinit`. Put `__myinit` in the initialization routine in the Linker preferences pane in the Project Settings dialog.

Listing 5.3 __myinit: Initializing the PPC compiler in C++

```
OSErr __myinit(InitBlockPtr initBlock)
{
  OSErr error;

  error=ProfilerInit(collectDetailed,
                  bestTimeBase, 20, 5);
  if (error == noErr)
    __sinit();
  return error;
}
```

Profiling MacApp

Before profiling a MacApp-based 68K project, go through the MacApp source code and comment out all calls to `UnloadSeg()`. There are only a few.

In a 68K project, calls to `UnloadSeg()` cause serious problems for the profiler. The profiler maintains pointers to locations in the code. If you call `UnloadSeg()` on a segment that has profiled routines,

that block of code is free to move around in memory, or be purged. At that time the profiler's internal pointers become invalid. This is the leading cause of "corrupted" profile data files.

Profilng Asynchronous Routines

The CodeWarrior profiler supports profiling asynchronous completion routines, VBL tasks, Time Manager tasks, deferred tasks, and related code. This is essentially transparent and a by-product of the internal structure and design of the profiler.

However, there are some things to be aware of when working with asynchronous routines and other code that runs at interrupt time.

- Don't try to call the profiler in a socket listener because the profiler adds too much latency for both LocalTalk and Ethernet devices.
- Be aware that running the profiler at interrupt time may cause problems because of the increased time that interrupts are disabled.
- Use the large code model for 68K code resources.

The small and smart code models for 68K can cause problems for completion routines. They use `A5` relative addressing to call the profiler. Because `A5` is not always set up when a completion routine is called, the embedded call to the profiler will not work. The large code model uses absolute addressing to call the profiler, which avoids the problem.

Another way around this problem is to have a small stub routine that sets up `A5` and calls your routine that does the real work. The stub routine should be compiled without a call to the profiler, and the real routine can call the profiler.

Profiling Abnormally Terminated Functions

The profiler correctly reports data for abnormally terminated functions that exited through the C++ exception handling model (try, throw, catch) or the ANSI C library `setjmp()` and `longjmp()` rou-

tines. You do not have to do anything to get this feature, it is automatic and part of the profiler's design.

However, there is a possibility of some errors in the reported results for an abnormally terminated function.

First, the profiler does not detect the abnormal termination until the next profiling call after the abnormal termination. Therefore, some additional time will be reported as belonging to the terminated function.

Second, if the next profiler event is a profiler entry, and the new stack frame for that function is larger than the frames that were abnormally exited, the profiler will not immediately detect that the original function was abnormally terminated. In that case the profiler will treat the function just entered as a child of the function abnormally terminated. The profiler will correct itself on the next profiling event without this property—that is, when the stack returns to a point smaller than it was when the abnormally terminated function exited.

Finally, remember that the profiler is not closed properly and the output file is not dumped when `ExitToShell()` is called. If you need to call `ExitToShell()` in the middle of your program and want the profiler output, call `ProfilerDump()`.

If you are using the profiler, you must always call `ProfilerTerm()` before `ExitToShell()`.

WARNING! If a program exits after calling `ProfilerInit()` without calling `ProfilerTerm()`, timers may be left running that could crash the machine.

Debugging Profiled Code

It is possible to debug code that has calls to the profiler in it. However, the profiler does interfere with stepping through code. You may find it simpler to debug non-profiled code, and profile separately. In this section, We'll take you through what happens when

you step into a profiled routine and step out of a profiled routine. In addition, we'll talk about the effect that stopping in the debugger has on the profile results.

See also the *CodeWarrior Debugger User Guide* for more information on how to use the debugger.

Stepping into a profiled routine

If you step into a profiled routine you'll see assembly code instead of source code. The compiler has added calls to `__PROFILE_ENTRY` at the start of the routine. This is how the profiler knows when to start counting time for the routine.

If you step through the assembly code far enough to get to the code derived from the original source code, then switch the view from source to assembly and back again, you can see the original source code.

Stepping out of a profiled routine

If you single-step out of a routine being profiled, you end up in the `__PROFILE_EXIT` assembly code from the profiler library. This is how the profiler knows when to stop counting time for the routine.

Effect of stopping on the profile results

If you stop in a profiled routine, the profiler counts all the time you spend in the debugger as time that routine was running. This skews the results.

WARNING! If you debug profiled code, you must be careful not to kill the code from the debugger. Remember, if you have called `ProfilerInit()` you must call `ProfilerTerm()` on exit or you may crash the computer.

(Mac OS) Profiler Limitations

For application programming, the CodeWarrior profiler has only one limitation. You must not call `UnloadSeg()` in a 68K application.

In a 68K project, calls to `UnloadSeg()` cause serious problems for the CodeWarrior profiler. The profiler maintains pointers to locations in the code. If you call `UnloadSeg()` on a segment that has profiled routines, that block of code is free to move around in memory, or be purged. At that time the profiler's internal pointers become invalid. This is the leading cause of "corrupted" profile data files.

See also Profiling Special Cases (Mac OS) for additional considerations in non-application projects.

(Mac OS) Profiler Tutorial

This tutorial is based on source code for a program called, Lyapunov, written by Andrea Pellizzon. Permission was obtained from the author to use her code in this tutorial. Metrowerks would like to thank Andrea for her generosity.

Lyapunov is a fractal generation program. This program was written based on information from the article, "The Lyapunov Space", written by A.K. Dewdney in "Le Scienze", issue 279. The program's name comes from a great Russian mathematician, Aleksandr M. Lyapunov. The math is explained in the manual in the Lyapunov 1.0.1 folder.

This tutorial was written and tested on a PowerPC 604/120 computer and a 33 MHz MC68030 computer running Mac OS 7.5.3 revision 2. The source code is written in Pascal. No attempt was made to convert the source to C or C++. Conversion is left as an exercise for the reader.

This tutorial will focus primarily on the PowerPC version of the program. If you have a 68K Mac OS computer, 68K versions of the project and compiled code are provided in the tutorial folders. The

procedures given here will work for both 68K and PowerPC versions of your programs.

This tutorial is broken down into the following sections:

- What You Need
- Getting Started
- Moving Ahead
- Now to Optimize
- One Last Time
- Congratulations!
- Where To Go From Here

What You Need

You'll find a folder called "CW Profiler Tutorial code" on the Reference CD. Copy this folder to your hard drive. This folder contains several other folders for various stages of the tutorial. The Lyapunov 1.0.1 folder contains the original Pascal source, documentation, binaries, and a QuickTime movie.

Getting Started

There are six main procedures followed in this tutorial. They are:

- Update for New Standards (If Required)
- Obtain Base Timings
- Profile Your Code
- Identify Problem Areas
- Decide On Your Goals
- Optimize Where Necessary

The last three steps are iterative. You repeat these until you achieve the performance gains you feel are required.

Update for New Standards (If Required)

The original source code for Lyapunov was written with an older version of Pascal. The code was brought up-to-date to compile with the latest Universal Pascal Interfaces. This also involved converting the floating point calls from using `SANE.p` to using `fp.p`.

TIP: SANE is only supported on 68K Mac OS computers. `fp.p` (and `fp.h`) is supported on both the 68K and PowerPC platforms.

The FPU version of the original had some code to predict what time the calculations would be finished. The conditional compilation directives were removed from that code so that it is included all the time. A routine to display the time in ticks was then added for convenience.

Obtain Base Timings

Now that the program is converted and running, you need to get an idea of how fast your code is running. This can be done in a variety of ways, from using a stop watch to modifying your code. In this example, the code was modified to display a dialog showing the number of ticks it took to run. You could also profile your code at this point to obtain timing information instead of putting up a dialog.

To make sure your optimizations are working, you need to run the same test case every time you run your code. This gives you a standard on which to base your optimizations. If you use a different test case every time, you will not know if your new code is working as easily.

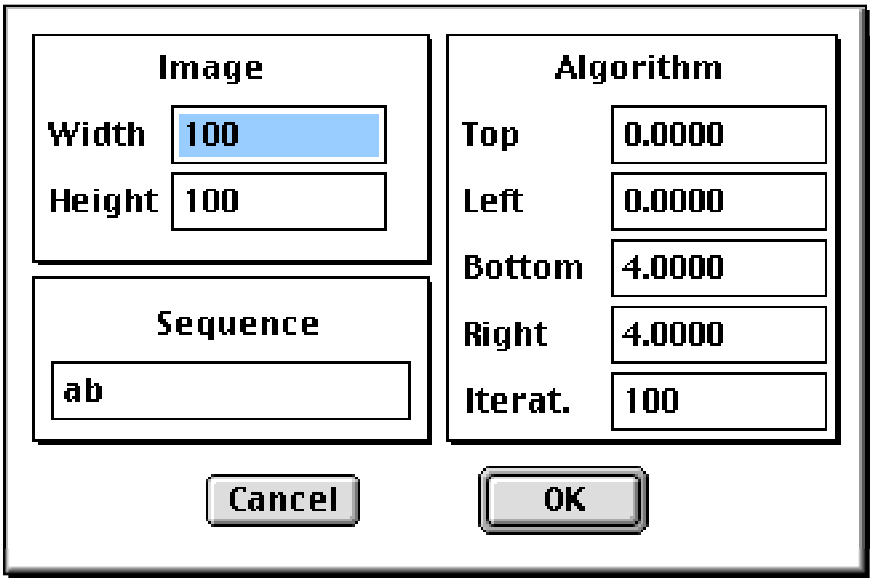
1. **Open the Solution Code 1 folder**
2. **Run one of the Lyapunov programs**

   Run the version of Lyapunov for the computer you are using. There are three targets: 68K (no FPU), 68K (with FPU), and PowerPC. This tutorial uses the PPC target settings. Substitute commands referring to this target with the one appropriate for your processor.

3. **Choose New from the File menu**

A dialog appears showing the program default parameters (Figure 5.1). These values are used for all the timings in this tutorial.

Figure 5.1 Parameters dialog



Using these values, you get the times shown in Table 5.7. These times were collected on routines compiled with no optimizations.

Table 5.7 Base Timings

Processor	Time (in ticks)
68K (non-FPU)	15310
68K (FPU)	2574
PowerPC	135

As you can see, the floating point performance of the PowerPC shines over the 68K.

Some programmers rely heavily on optimizing compilers, which can give a performance boost. However, this is no substitute for

clean, fast code. To prove this, try a test with all the optimizations turned on.

1. **Open Lyapunov.mcp**

   You can use either the Solution Code project or the project file from the Exercise 1 folder.

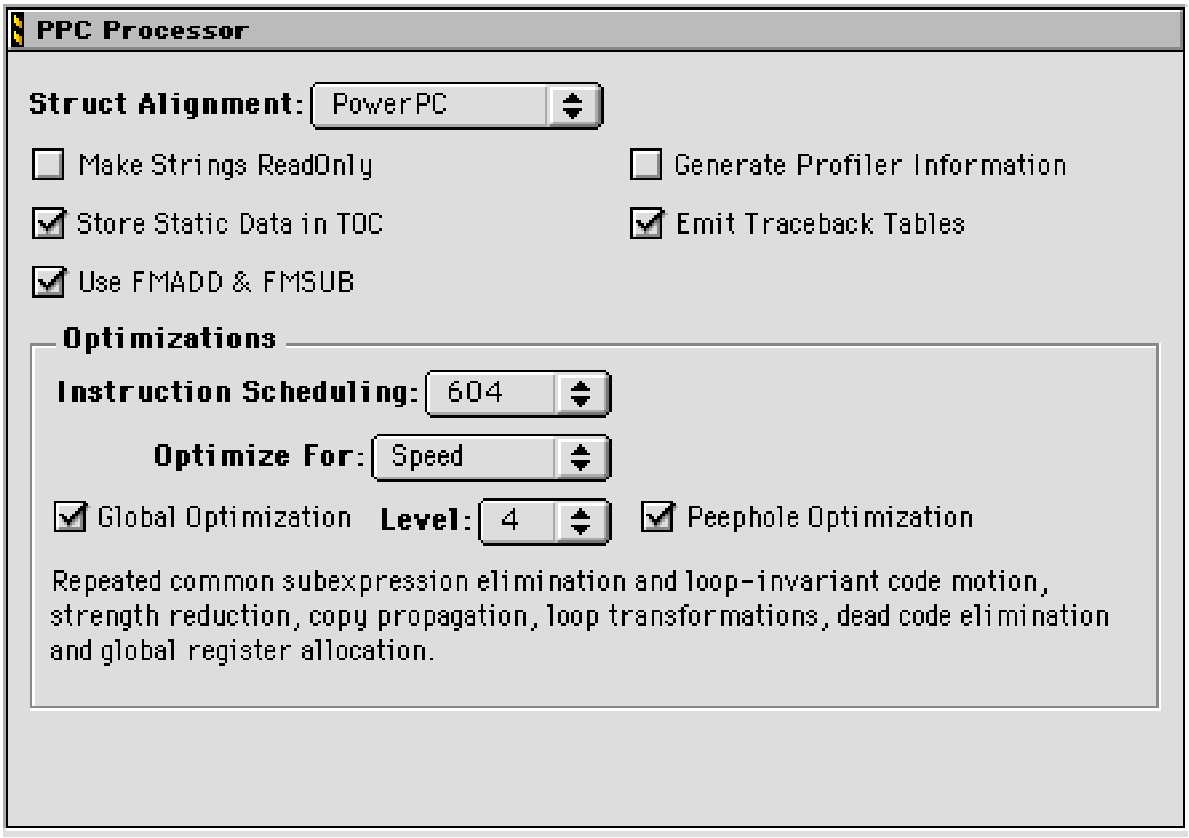
2. **Choose LyapunovPPC Settings... from the Edit menu**

3. **Choose PPC Processor from the list (Figure 5.2).**

4. **Turn on all optimizations**

   Set **Instruction Scheduling** to 604 (or whatever PowerPC processor you have), **Peephole Optimization** on, and **Global Optimization** to level 4. The resulting dialog is shown in Figure 5.2.

Figure 5.2 PPC optimizer settings


PPC Processor
Struct Alignment: PowerPC
Make Strings ReadOnly
Generate Profiler Information
Store Static Data in TOC
Emit Traceback Tables
Use FMADD & FMSUB
Optimizations
Instruction Scheduling: 604
Optimize For: Speed
Global Optimization
Level: 4
Peephole Optimization
Repeated common subexpression elimination and loop-invariant code motion, strength reduction, copy propagation, loop transformations, dead code elimination and global register allocation.

5. **Compile and Run**

   The resulting time is only 130 ticks. A 5 tick difference is not such a huge boost.

6. **Turn off all the optimizations**

   The next step is to use the profiler to find out where the time is going and see what can be done about it.

Profile Your Code

For this part of the process, you need to add the proper profiler library and source code. Then, modify your code to use the profiler. Set the project settings to output profiler calls. Finally, run your code and view the timing results in the MW Profiler application. If your program is really big, you may have to increase the memory allocation to use the profiling calls.

Profile Exercise 1

1. **Open the Lyapunov project from the Exercise 1 folder**

   Choose the appropriate project file for your computer.

2. **Add the correct profiler library and source code**

   CodeWarrior displays the Target Designation dialog box (Figure 5.3). Make sure you choose the correct target for your system configuration. The `ProfilerPPC.Lib` is shown in Figure 5.4.

Figure 5.3 Target designation dialog box

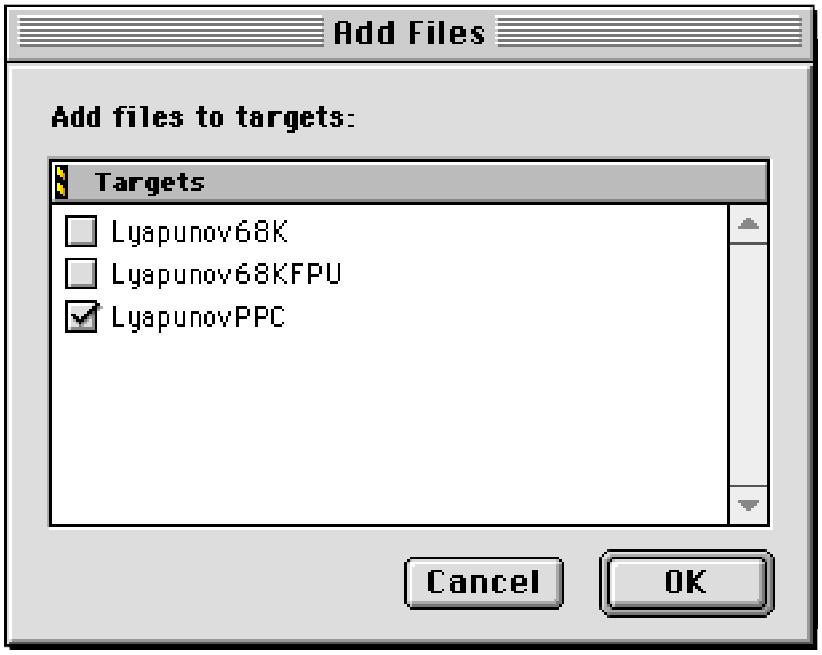
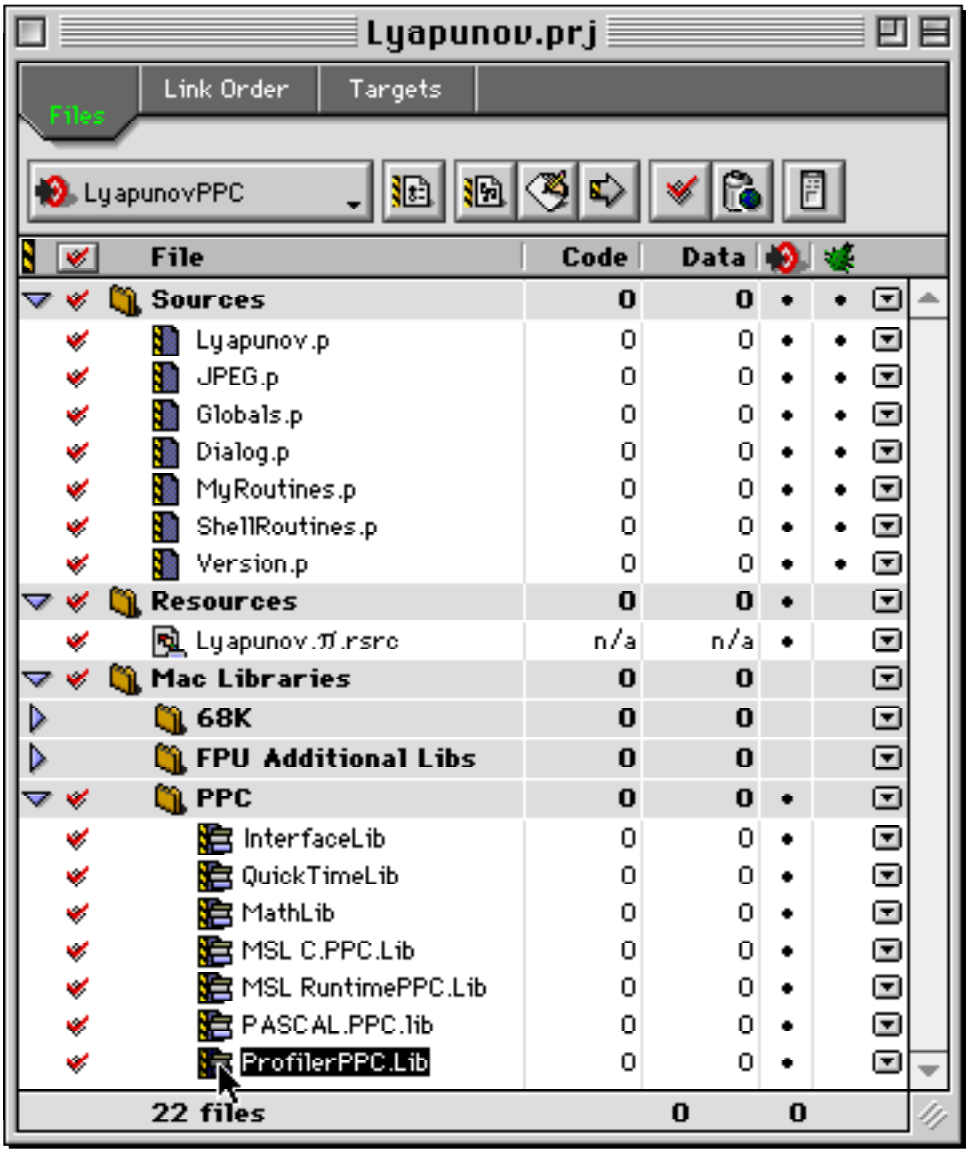


Figure 5.4 Adding profiler library



The proper profiler library depends on the kind of program you are writing and the platform you are targeting. The profiler libraries are listed in Table 5.8 below.

Table 5.8 Profiler libraries

Library	Processor	When To Use
`profiler68k(Small).lib`	68K (A5 globals)	near/small model applications
`profiler68k(Large).lib`	68K (A5 globals)	far/large/smart model applications
`profiler68kA4(Small).lib`	68K (A4 globals)	near/small model code resources
`profiler68kA4(Large).lib`	68K (A4 globals)	far/large/smart model code resources
`Profiler68kCFM.lib`	68K (A5 globals)	code fragments and shared libraries
`ProfilerLIB`	CFM 68K & PowerPC	code fragments and shared libraries
`profilerPPC.lib`	PowerPC	PowerPC applications and shared libraries

TIP: Use `ProfilerLIB` when you want to collect information across shared library boundaries. If all you want is data from one library, use one of the static link libraries and link it into your project.

3. **Add source code files**

Add `profiler.p` as shown in Figure 5.5. The same file is used for 68K and PowerPC platforms. Make sure all targets are selected in the Target Designation dialog box.

NOTE: If you are using C, you include the `profiler.h` header file in your main program file. The same libraries are used for Pascal, C, and C++/PowerPlant.

Figure 5.5 Adding Profiler.p

4. Modify your code to include calls to the profiler
`Lyapunov.p`

It is always best to profile your entire application the first time through. This will give you times for every routine in your program. You can then use this information to see where improvements are needed most.

a. Open Lyapunov.p from the project window

b. Add profiler to your `uses` statement

Your `uses` line should look like the following:

```
uses
  QDOffScreen, Notification, Types, QuickDraw,
  Menus, OSUtils, ToolUtils, Windows, Events,
  DiskInit, Dialogs, Globals, ShellRoutines,
  MyRoutines, Profiler;
```

c. Add calls to `ProfilerInit()` in your main procedure

Use the function popup menu to go to the top of your main program. Add a call to `ProfilerInit()` at the very top of this procedure. Surround the call to `ProfilerInit()` with conditional compiler directives. This way, you can run your program without the profiler for testing. You don't have to keep adding and deleting, or commenting and uncommenting your code. The solution code is given to you here with code above and below `ProfilerInit()` to show you where things go.

```
  var
    time: longint;
    err:  OSErr;
begin
{$IFC OPTION(PROF)}
  if (ProfilerInit(collectDetailed,
    bestTimebase, 100, 20) = noErr) then begin
{$ENDC}
    InitGraf(@qd.thePort);
    InitWindows;
    InitMenus;
```

d. Add calls to `ProfilerTerm()` and `ProfilerDump()`.

Now you need to add the profiler calls that create the data file and end the profile session. Remember, you are profiling the entire application. Therefore, put these calls at the end of your program. When you quit your application, the profiler data file will be created. The code you need is shown below, again with surrounding code so you know where things are.

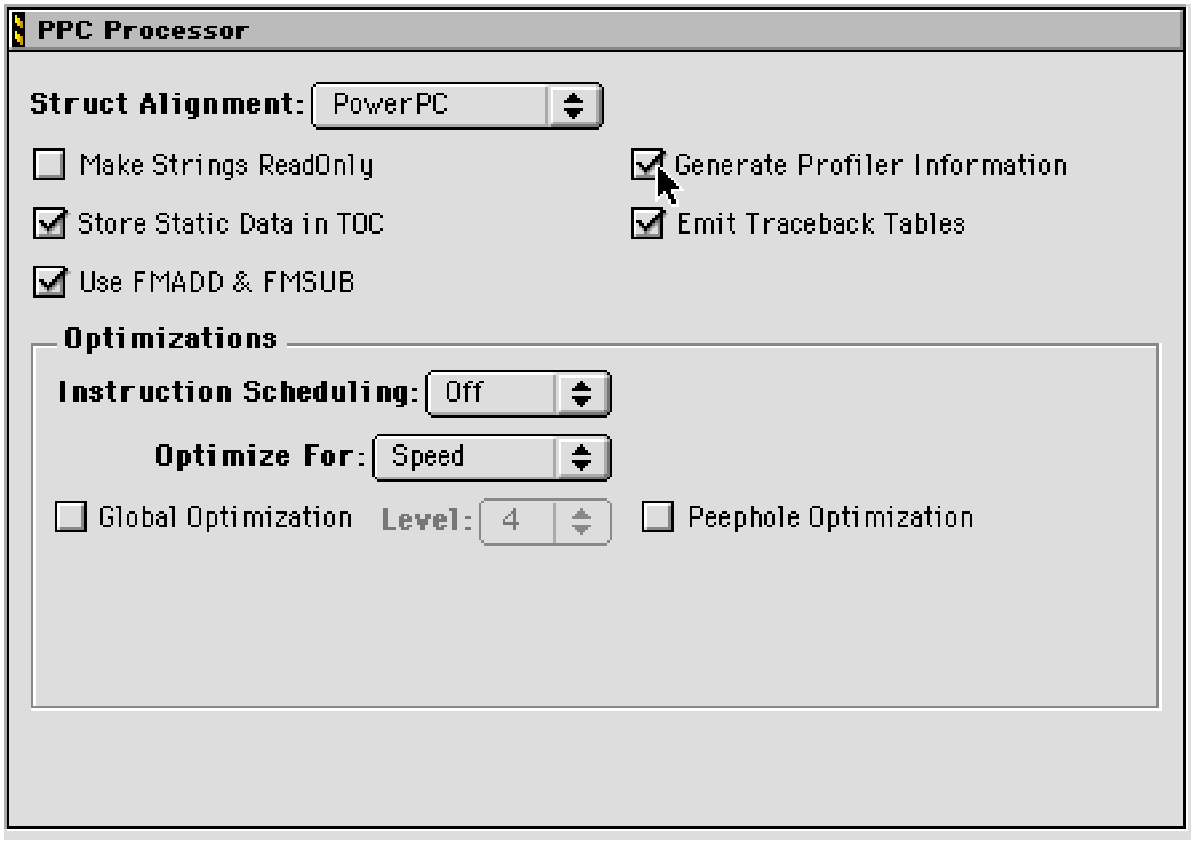
```
      if running then
        DoCalc;
    until quit;
{$IFC OPTION(PROF)}
    err := ProfilerDump('Lyapunov');
    ProfilerTerm;
```

```
  end;
{$ENDC}
end.
```

5. **Set the project settings**

   Turn on **Generate Profiler Information** in the **PPC Processor** preference panel as shown in Figure 5.6.

Figure 5.6 The project settings



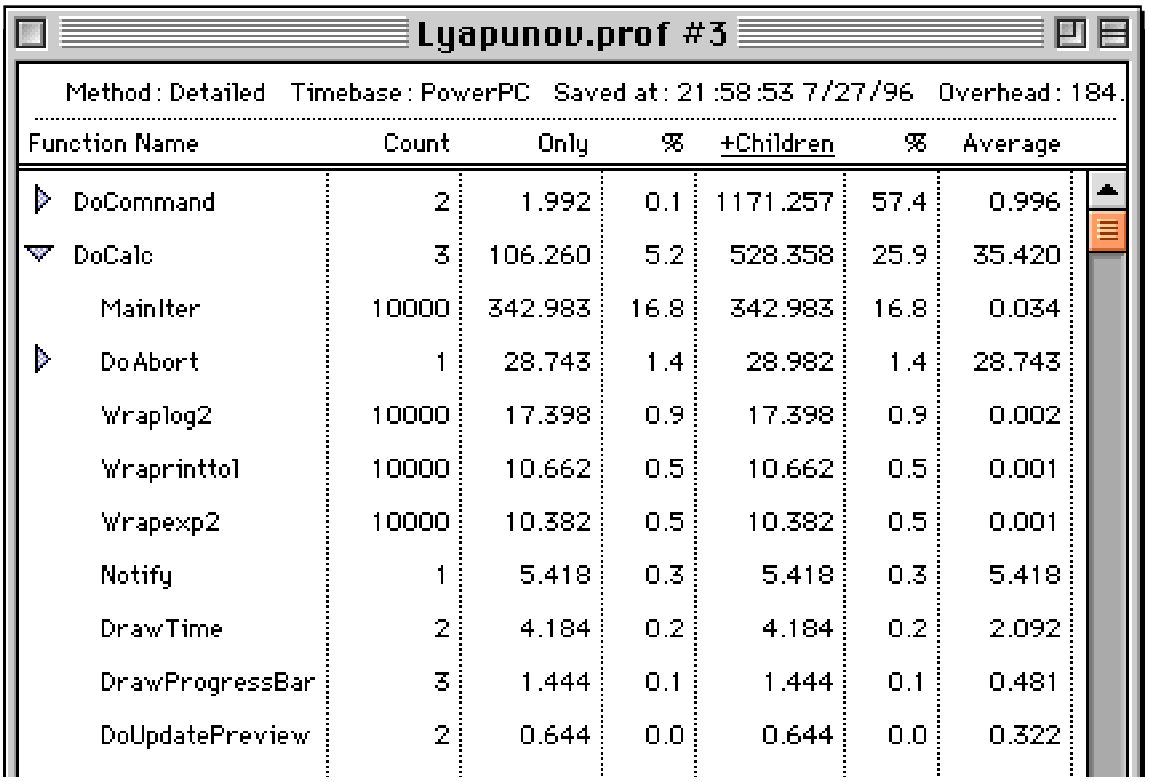
6. **Run your code and view the results**

   Compile and run your application. Use the default values provided by the program. When the program is finished, choose **Quit** from the **File** menu.

Identify Problem Areas

At this point, you should have a MW Profiler data file called "Lyapunov" in the same folder as your application. Open this file with the MW Profiler viewer application. Choose **by +Children** from the **View** menu. The results are shown in Figure 5.7.

Figure 5.7 First profile



Function Name	Count	Only	%	+Children	%	Average
DoCommand	2	1.992	0.1	1171.257	57.4	0.996
DoCalc	3	106.260	5.2	528.358	25.9	35.420
MainIter	10000	342.983	16.8	342.983	16.8	0.034
DoAbort	1	28.743	1.4	28.982	1.4	28.743
Wraplog2	10000	17.398	0.9	17.398	0.9	0.002
Wraprinttol	10000	10.662	0.5	10.662	0.5	0.001
Wrapexp2	10000	10.382	0.5	10.382	0.5	0.001
Notify	1	5.418	0.3	5.418	0.3	5.418
DrawTime	2	4.184	0.2	4.184	0.2	2.092
DrawProgressBar	3	1.444	0.1	1.444	0.1	0.481
DoUpdatePreview	2	0.644	0.0	0.644	0.0	0.322

TIP: The results shown in this tutorial may be different than what you see on your screen. The timings depend on the processor you are using. If you want to see the same results shown in this tutorial, the data files for each profile can be found in the Profile Data folder.

Notice there are two routines taking most the time, `DoCalc()` and `DoCommand()`. `DoCommand()` calls `DoNew()`, which displays a

modal dialog asking for the parameters to graph. All the time spent in this routine is time waiting for you to click the **OK** button. Nothing can be done with this routine.

`DoCalc()`, on the other hand, is something you can make faster as there is no user input. `DoCalc()` is responsible for calculating the graphic image. The rest of this tutorial concentrates on speeding up this calculation.

Notice also that `MainIter()` is called 10,000 times. Any speed gain achieved in `DoCalc()` will be magnified 10,000 times!

The calculation is at the bottom of the file, MyRoutines.p. The main routine is `DoCalc()`, but there are some other routines used also.

Decide On Your Goals

At some point, you need to decide on what goals you want to achieve. You now know how the program is performing. You know where the problems are. You need to decide what kind of performance gains you want in your program.

In this case, speeding up the calculation routines to the point where they are no longer annoying is a good start. You have to wait a long time, even on the PowerPC version, for the calculation to finish and display the resulting image.

Optimize Where Necessary

You know where the optimizations are needed. The question is, "How can this code be optimized?" It may not be easy, at this point, to see how to optimize the code. So, further analysis is needed. Read on!

Moving Ahead

This section of the tutorial concentrates on taking a closer look at the problem code. This can narrow the scope enough to make the task of optimizing easier and achievable.

The following topics are discussed:

- A Closer Look
- Creating Wrapper Routines
- Modify Your Code
- The Results

A Closer Look

Before you can optimize the `DoCalc()` routine, it's important to note that `DoCalc()` calls a few other routines. Notably, there are calls to the math library routines `exp2()`, `log2()`, `rinttol()`, and calls to the Mac OS Toolbox routines `SetGWorld()`, and `SetCPixel()`.

If one of your routines is taking up a lot of time, and makes Mac OS Toolbox or library calls you don't have code for, it is best to separate these calls from your routines by creating *wrapper* routines. You might think there is nothing you can do about the Toolbox, but sometimes there is. Read on!

Creating Wrapper Routines

The profiler doesn't give any detail on the Toolbox or compiled library routines that it calls in a routine, so you can wrap some of the routines that are called in `DoCalc()` to get profile information on them.

A wrapper is a routine you create that has one purpose, that is to call one specific routine. In this case, the wrapper routine will call the Mac OS Toolbox and math library routines. For example, the wrapper routine for `exp2()` is shown in Listing 5.4.

The wrapper routine will do two things for you. First, it separates the library routines from your own. Second, it allows you to effectively profile the library routine. Profile data for your wrapper routine will show up in the viewer application.

Listing 5.4 Wrapping a Toolbox routine

```
FUNCTION Wrapexp2(a: Extended) : Extended;
BEGIN
```

```
  Wrapexp2 := exp2(a);
END;
```

Notice that parameters are kept consistent with the library routine. Note also the addition of the word, `Wrap`, added to the routine. This will help you know which routines you've written wrappers for. Once you've written the wrapper routines, you replace the Toolbox or library call with the call to your routine. For example, the call to the `Wrapexp2()` function would look like this:

```
temp := Wrapexp2(Wraplog2(lyap) / Iter + c3);
```

TIP: The Pre-release folder contains versions of the Mac OS ToolBox libraries with Profiler Calls turned on. This library can be used to acquire profile times for these routines. However, as a general rule, these libraries would not be available. The techniques taught here can apply to any library for which you do not have source code and is the better procedure to use.

Modify Your Code

At this stage, you'll want to modify your code to include wrapper routines for the following routines: `exp2()`, `log2()`, `rinttol()`, `SetGWorld()`, and `SetCPixel()`.

Profile Exercise 2

1. **Write the wrapper routines**
   `MyRoutines.p`

   Write wrapper routines for `exp2()`, `log2()`, `rinttol()`, `SetGWorld()`, and `SetCPixel()`. Place these new routines above `DoCalc()`. The solution code is shown in Listing 5.5.

Listing 5.5 Wrapper routines

```
function Wrapexp2(a: Extended) : Extended;
begin
  Wrapexp2 := exp2(a);
end;
```

```
function Wraplog2(a: Extended) : Extended;
begin
  Wraplog2 := log2(a);
end;

function Wraprinttol(a: Extended) : LongInt;
begin
  Wraprinttol := rinttol(a);
end;

procedure WrapSetGWorld(port : CGrafPtr;
               gdh : GDHandle);
begin
  SetGWorld(port, gdh);
end;

procedure WrapSetCPixel(a1, b1: integer;
               cPix: RGBColor );
    begin
        SetCPixel(a1, b1, MyColorTable[color]);
    end;
```

2. Replace Toolbox routines with calls to your wrappers

`DoCalc()` `MyRoutines.p`

Now, replace the calls to the Toolbox, with calls to your wrapper routines. The code is shown below. Existing code is shown in italics.

```
begin
  coeff := TickCount + TicksPerLoop;
  while TickCount < coeff do
  begin {c1,c2,c3 calcolati in DoNew}
    a := MinX + a1 * c1;
    b := MinY + b1 * c2;
    lyap := MainIter(a, b, Iter);
{color := round(Exp2(log2(lyap) / Iter + c3));}
    temp := Wrapexp2(Wraplog2(lyap) / Iter +
c3);
    color := Wraprinttol(temp);
    if color > 255 then
```

```
            color := 0;

        WrapSetGWorld(PictGW, nil);
        WrapSetCPixel(a1, b1);
        WrapSetGWorld(GWorldPtr(theWind),
    GetMainDevice);
        WrapSetCPixel(a1, b1, MyColorTable[color]);

        Counter := Counter + 1;
        a1 := a1 + 1;
```

3. **Profile code again**

   Compile and run the code again. A new data file called "Lyapunov2" is created in the same folder as the application.

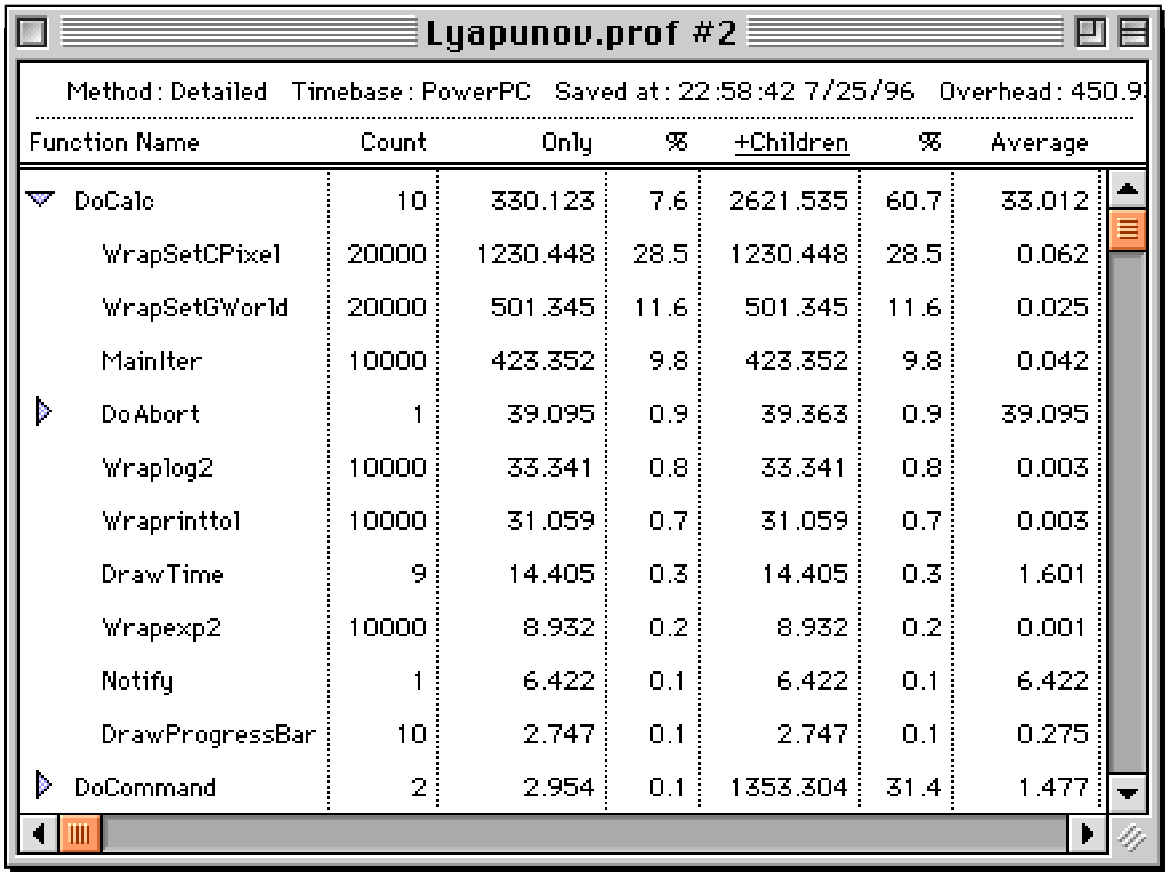
Why Lyapunov2?

You might be wondering why the data file is called "Lyapunov2" when you didn't make any changes to the profiler routines. As long as you don't change where you put the profiling routine calls, each run through the profiler will produce a sequential data file. This allows you easily track your changes and compare them to make sure your new code is working the way you think it should.

The Results

The profile data is shown in Figure 5.8. The results are very interesting. They show that the graphics routines, `SetCPixel()` and `SetGWorld()`, are the culprits currently, not the calculations!

Figure 5.8 Second Profile



Now to Optimize

Up to now, you've learned how to use the profiler to spot problem areas. You've also learned how to create wrapper routines for the Mac OS Toolbox and any compiled library, thus, effectively profiling those routines. Now you've come to point where you need to make a decision. Try and optimize the code, or leave it as is.

This section focuses on optimizing DoCalc(). Topics discussed are:

- Optimize the Toolbox?
- Inlining Routines
- What's Happening?

- More Results

Optimize the Toolbox?

You can't optimize the Toolbox. You might be able to make a toolbox routine go faster by changing the parameters to a routine (like passing in a shorter string to `StrLower()`, if you only need part of it) or by calling it fewer times.

In this case you see that `SetCPixel()` takes up a large amount of time to do its magic. You can save a lot of time if you replace it with your own, more efficient routine to solve this specific problem. It is important to note that this wouldn't work in every application.

Note the number of times `WrapSetCPixel()` is called. You can fix that by getting rid of `SetCPixel()` and writing your own version. As the programmer of your application, you are working with many "knowns". You know some things that `SetCPixel()` has to figure out each time it's called. You know that the offscreen bitmap will always be 8 bits per pixel, because you've created it. There is one problem with `SetCPixel()`, it has to do color mapping. `SetCPixel()` takes an index value and maps it to an `RGBColor`. This is a time-consuming process. Note that it is passed an `RGBColor`, which it maps to a color index. You can store the color indexes directly and save time. By writing our own `SetCPixel()`, you can also get rid of the `SetGWorld()` call for the offscreen bitmap, saving even more time. You can take out 10,000 calls to `SetGWorld()` and another 10,000 calls because each pixel is no longer being drawn to the screen. A grand total of 20,000 calls to `SetGWorld()` have been eliminated!

The current code always draws the pixel to the window even if it's not visible. That's something that's easy to fix. It also draws each pixel individually. You can change this to draw only once at the end of each `DoCalc()`. This will make the drawing seem less smooth, but it will save nearly 40% of the time!

Inlining Routines

There is one other thing you should do. You should put the replacement `SetCPixel()` code directly into `DoCalc()`. This is called *in-*

lining. This will save a lot of time and a lot of extraneous calls to `LockPixels()` and `UnlockPixels()`.

Profile Exercise 3

The new `DoCalc()` procedure is shown in Listing 5.6. The solution code can also be found in the Solution Code 3 folder. Existing code is shown in italics.

Listing 5.6 New DoCalc procedure in MyRoutines.p

```
procedure DoCalc;
  var
    a, b, lyap, temp: Extended;
    coeff: Longint;
    base: Ptr;
    aPixel: Ptr;
    pixMap: PixMapHandle;
    rowBytes: longint;
  begin
    coeff := TickCount + TicksPerLoop;
    pixMap := GetGWorldPixMap(PictGW);
    if LockPixels(pixMap) then begin
      base := GetPixBaseAddr(pixMap);
      rowBytes := BAND(pixMap^^.rowBytes, $3FFF);
      aPixel := Ptr(longint(base) + b1 * rowBytes + a1);
      while TickCount < coeff do
      begin {c1,c2,c3 calcolated in DoNew}
        a := MinX + a1 * c1;
        b := MinY + b1 * c2;
        lyap := MainIter(a, b, Iter);
  {color := round(Exp2(log2(lyap) / Iter + c3));}
        temp := Wrapexp2(Wraplog2(lyap) / Iter + c3);
        color := Wraprinttol(temp);
        if color > 255 then
          color := 0;
        aPixel^ := 255 - color;
        aPixel := Ptr(longint(aPixel) + 1);
        Counter := Counter + 1;
        a1 := a1 + 1;
        if a1 = SizeX then
```

```
      begin
        a1 := 0;
        b1 := b1 + 1;
        aPixel := Ptr(longint(base) + b1 * rowBytes);
        if b1 = SizeY then
        begin
          DrawProgressBar;
          Notify;
          DoAbort(False);
          Exit(DoCalc);
        end;
      end;
    end;
    if Counter >= NextPerCent then
      DrawProgressBar;
    if Counter >= NextTime then
      DrawTime;
    DoUpdatePreview;
    UnlockPixels(pixMap);
  end; {LockPixels}
end;
```

What's Happening?

`SetCPixel()` has been replaced with some explicit code that understands the layout of a `PixMap` in memory. Calls to `LockPixels()` and `UnlockPixels()` are necessary to access the GWorld directly without going through `SetCPixel()`. These calls are moved outside of the loop, reducing the number of times they are called.

Strength reduction

An optimization technique, called strength reduction, has been used on the `aPixel` variable. Strength reduction means you replace an operation in a loop with one that is cheaper. Generally, you are replacing an absolute calculation with an incremental calculation. An example in C would be using a pointer to index through an array instead of using the normal array operations. For example:

```
int  a[100];
int  i, total = 0;
for (i = 0; i < 100; i++)
    total += a[i];
```

could be replaced with

```
int  a[100];
int  i, total = 0;
int  *p;
for (i = 0, p = a; i < 100; i++)
    total += *p++;
```

The important part about the strength reduction on the `aPixel` variable is to notice that `aPixel` is calculated differently depending on where in the image it is. A PixMap is a series of rows, each row is `(pixMap^^.rowBytes)` bytes long. If your bitmap is 100 pixels wide, and each pixel is one byte (because it's an 8-bit deep PixMap), `rowBytes` may not be 100. Therefore, you have to do the multiply to calculate the start of each row. But, within a row, the bytes are adjacent, so you can be cheap and get by with adding 1.

No direct-to-screen drawing

The new code does not do any direct-to-screen drawing, only to the offscreen GWorld. The depth of the GWorld is known, but the resolution of the user's screen (or screens) are not. `CopyBits()` is still used (in the `DoUpdatePreview()` routine) to draw from the offscreen GWorld to the screen. However, large groups of pixels are copied efficiently. `CopyBits()` is only called when needed (approximately every 20 ticks). This makes the screen drawing a little more jerky, but speeds up the overall calculation. On the PowerPC, it is fast enough that you don't see any jerkyness at all!

More Results

After making this change, recompile without **Emit Profiler Calls** turned on in the project settings. Run the PowerPC version. The run time dropped to 32 ticks! That's pretty good. 75% of the calculation time was being wasted drawing into the offscreen bitmap (and the window when it was open).

One Last Time

Remember, profiling is an iterative process. You continue to profile and optimize until you're satisfied with the results. It's completely up to you when you are satisfied. For this tutorial, one more profile is required.

This section focuses on squeezing out the last bit of performance possible. Topics include:

- Looking for More
- Oh No! Not Assembly!
- Final Times

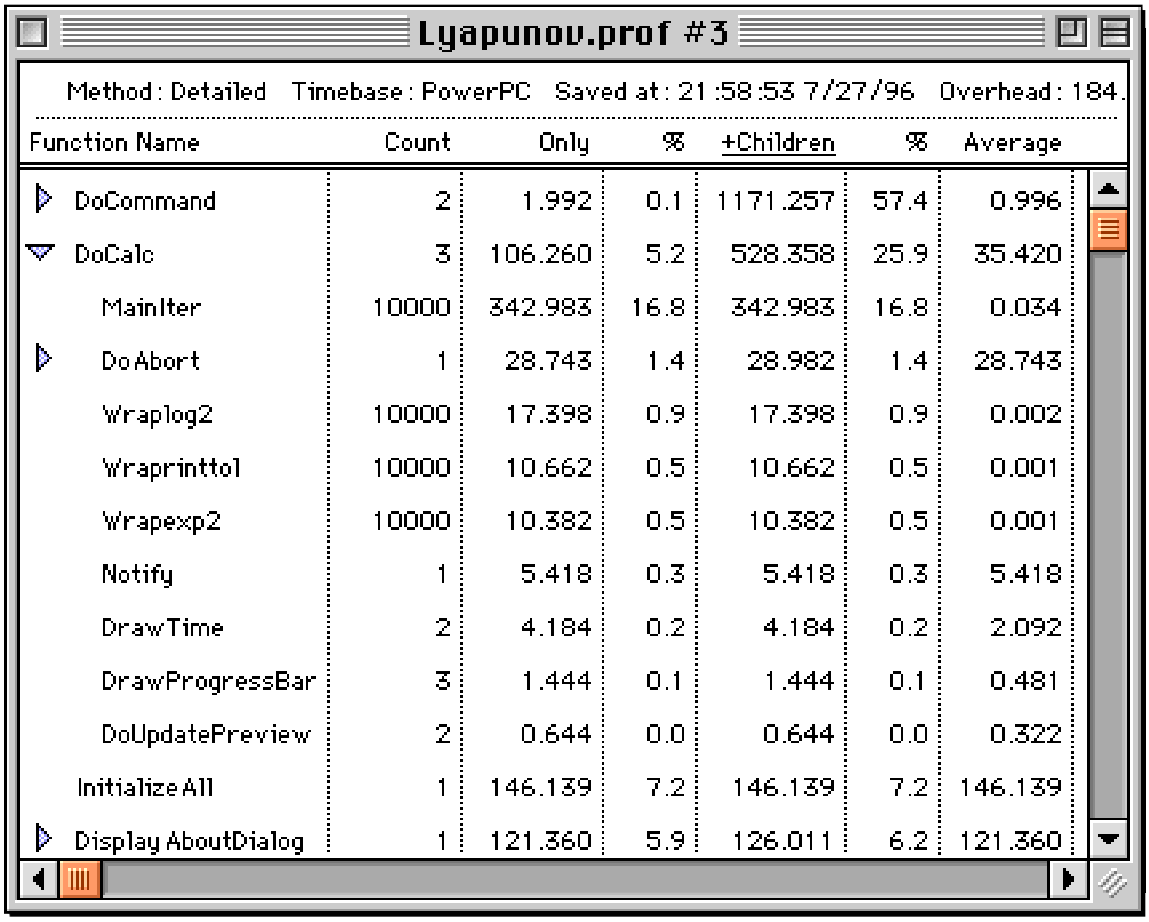
Now, recompile with **Emit Profiler Calls** turned on to find out what's next to change. The results are shown in Figure 5.9. You see that `MainIter()` is taking 64.8% of the time. This is calculated by dividing the percentage time of `MainIter()` by the percentage time of the parent routine, `DoCalc()` (16.8/25.9 = 64.8%).

Looking for More

If you look at `MainIter()` carefully, you'll notice that the calculation `x:= exp(1.0) / 10` is being done every time through the function. The value of `x` never changes. The first change is to make the calculation of `exp(1.0) / 10` once and store it in a global, reloading it when necessary.

TIP: You should look for these kinds of calculations involving constants and take them out of any loops. Calculate them ahead of time, store them in a global, and read it only when necessary.

Figure 5.9 Final profile



Lyapunov.prof #3

Method: Detailed Timebase: PowerPC Saved at: 21:58:53 7/27/96 Overhead: 184.

Function Name	Count	Only	%	+Children	%	Average
DoCommand	2	1.992	0.1	1171.257	57.4	0.996
DoCalc	3	106.260	5.2	528.358	25.9	35.420
MainIter	10000	342.983	16.8	342.983	16.8	0.034
DoAbort	1	28.743	1.4	28.982	1.4	28.743
Wraplog2	10000	17.398	0.9	17.398	0.9	0.002
Wraprinttol	10000	10.662	0.5	10.662	0.5	0.001
Wrapexp2	10000	10.382	0.5	10.382	0.5	0.001
Notify	1	5.418	0.3	5.418	0.3	5.418
DrawTime	2	4.184	0.2	4.184	0.2	2.092
DrawProgressBar	3	1.444	0.1	1.444	0.1	0.481
DoUpdatePreview	2	0.644	0.0	0.644	0.0	0.322
InitializeAll	1	146.139	7.2	146.139	7.2	146.139
DisplayAboutDialog	1	121.360	5.9	126.011	6.2	121.360

Oh No! Not Assembly!

To squeeze out that last bit of performance from an application generally requires a knowledge of the assembly language of the processor, and the timing requirements of the instructions. Just knowing that integer division and multiplication are slower on the PowerPC can be helpful. Floating point division and multiplication are relatively much faster than their integer versions, taking only 2 cycles for double precision, and 1 cycle for single precision. This information can be obtained from the chip manufacturer.

You can perform a disassembly of `MainIter()` to find out what is going on at the machine level and see if it can be changed or opti-

mized further. You can use either MacsBug or the CodeWarrior IDE to perform the disassembly. The CodeWarrior IDE method is shown here.

Profile Exercise 4-Part 1

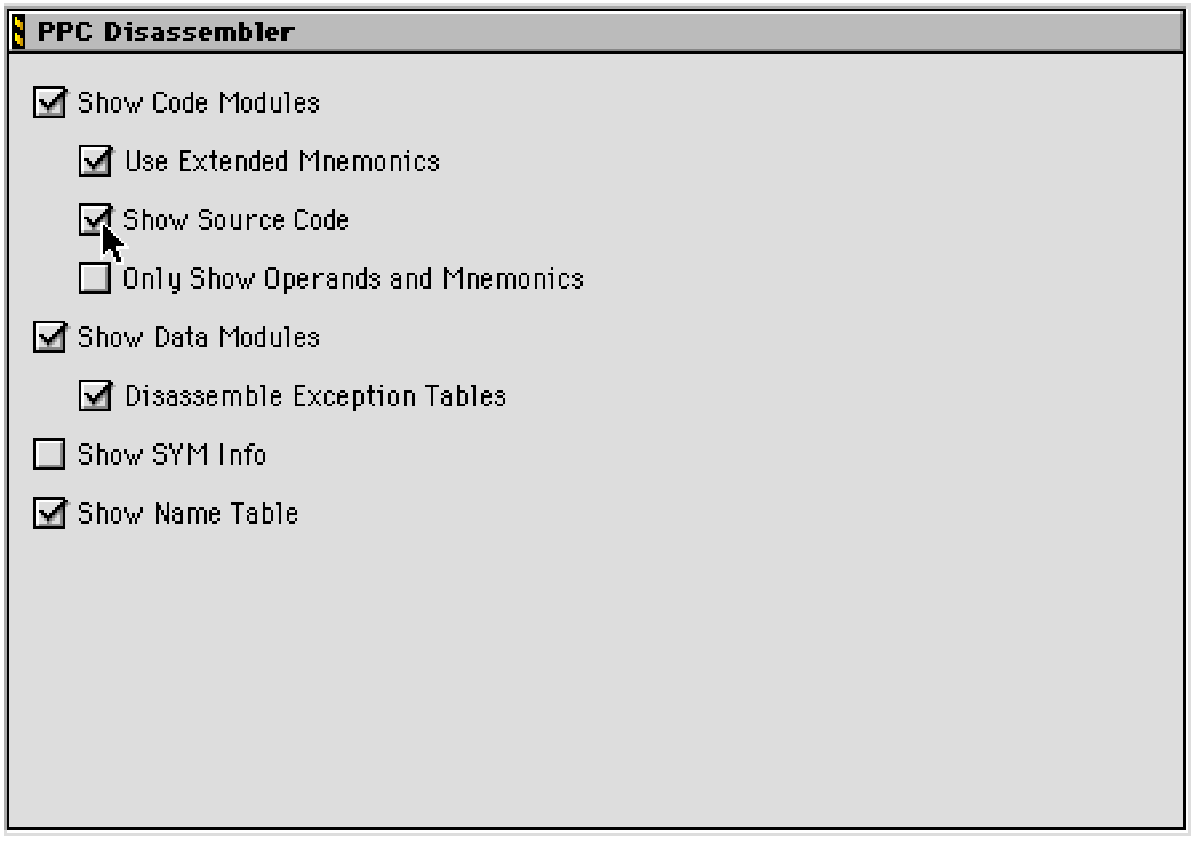
1. **Make sure project is up to date**

   Choose **Bring Up To Date** from the **Project** menu.

2. **Choose disassembly preferences**

   Choose **Show Source Code** from the **PPC Disassembler** preference panel as shown in Figure 5.10. The option is the same in the **68K Disassembler** preference panel.

Figure 5.10 Disassemble preferences



3. **Choose file to disassemble**

   Click on the `MyRoutines.p` file from the project window.

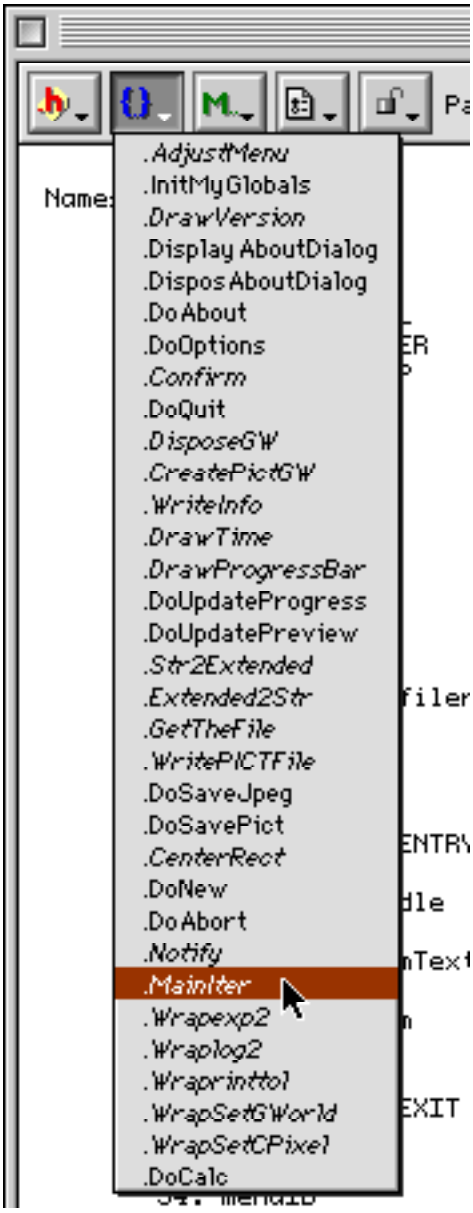
4. **Dissassemble**

   Choose **Dissassemble** from the **Project** menu. You will get a new file called MyRoutines.p.dump.

5. **Choose routine to look at**

   Choose .**MainIter** from the IDE's function menu (Figure 5.11).

Figure 5.11 Choosing the disassembled routine



A disassembly of `MainIter()` shows that the mod operator is implemented by this formula:

```
a mod b = a - (a div b) * b
```

This can be seen from the assembly code in Listing 5.7. The calculation is shown in italics.

Listing 5.7 Disassembly of `MainIter()`

```
0000005C: 48000070  b        *+112  ; $000000CC
      if occ[n mod occLen] then
      begin
00000060: A89F0000  lha      r4,0(r31)
00000064: 7CC50734  extsh    r5,r6
00000068: 7C6523D6  divw     r3,r5,r4
0000006C: 7C6321D6  mullw    r3,r3,r4
00000070: 7C632850  sub      r3,r5,r3
00000074: 7C7E18AE  lbzx     r3,r30,r3
00000078: 2C030000  cmpwi    r3,0
0000007C: 41820024  beq      *+36 ; $000000A0
```

Register r4 contains `accLen`, r5 contains `n`. The `divw`, `mullw`, and `sub` calls are used to implement the mod operator. Finally, the `lbzx` call is the array lookup and the `beq` call is the branch to start the loop over again.

Unfortunately on the PowerPC, integer division is not very fast. The average cycle count is around 30 cycles. Strength reduction can be used again to reduce this to an addition, and a test for the upper bound. This change now costs 3 cycles instead of 30.

Now go back and modify your code.

Profile Exercise 4-Part 2

1. **Move the calculation out of `MainIter()`**

   Move the calculation out of the `Mainiter()` routine and place it into `InitMyGlobals()` as shown below. Existing code is in italics.

```
for i := 0 to 255 do
      with MyColorTable[i] do
      begin
        red := BSL(i, 8);
```

```
        green := BSL(i, 8);
        blue := BSL(i, 8);
      end;

    initialX := exp(1.0) / 10;
    LtBluePixPat := GetPixPat(128);
```

2. Declare `initialX` in the `var` section of the Globals.p file

`initialX` is a global variable. Move the declaration to Globals.p. Notice in the `MainIter()` routine, `x` was declared as `Extended`.

```
startTicks, endTicks : Longint;
  tickString: Str255;
  initialX : Extended;
```

3. Modify `MainIter()`

Now you need to modify `MainIter()` to rework the mod operator.

```
function MainIter (a, b: extended;
                   Iter: Integer): Extended;

  var
    n: Integer;
    index: Integer;
    tot, x: Extended; {This declaration stays}
  begin
    tot := 1.0;
{x := 2.0; Not so good !!!}
{if the first occurence is 25/8: x <- r * 0.2 *
0.8 = 0.5}
{so derivative r(1-2x) will be *exactly* 0}
    x := initialX;
    index := 1;
    for n := 1 to Iter do
    begin
      if occ[index] then {n mod occLen}
      begin
        x := a * x * (1.0 - x);
        tot := tot * a * (1.0 - x - x);
      end
      else
      begin
```

```
            x := b * x * (1.0 - x);
            tot := tot * b * (1.0 - x - x);
          end;
          index := index + 1;
          if (index = occLen) then
            index := 0;
        end;
        MainIter := abs(tot);
    end;
```

4. **Compile and test**

Final Times

Compiling without Profiler Calls on shows that the time is now:

Table 5.9 Final timings

Processor	Time (in ticks)
68K (non-FPU)	12451
68K (FPU)	1404
PowerPC	19

The PowerPC version now only takes 19 ticks! A calculation that once took 2 seconds, can now be done 4 times per second! That's a significant speed boost! Now that the code is optimized, you can use the compiler optimizations CodeWarrior gives you to gain a little more speed (down to 16 ticks).

Congratulations!

You've successfully turned a slow program into a speed demon! You should now have a better understanding of how the profiler can be used to find problem areas in your code. You should also have some knowledge of optimizing techniques and problem areas to look for in your code.

Where To Go From Here

Can anything else be done to improve the performance? It may be possible to move all the floating point code to single precision. However, an initial test revealed that this modification didn't change the time. Another thing to try is to inline `MainIter()` into `DoCalc()`. But the prologue and epilogue code is now quite small, so the savings would be fairly small.

Add Features

Now that you have the code working much faster than the original, you can spend some time adding more features. There is a QuickTime movie in the Lyapunov 1.0.1 folder. It was created by doing several runs of the program and saving the resulting picture to a file. These pictures were then placed individually into a QuickTime movie. It would be a really cool feature to add an option that can do all of this with a single run of the application. You could write each picture directly to a QuickTime movie file. A feature like this would not have been able to be added with the original code.

Good luck and happy profiling!

6 Profiling PlayStation Code

This reference section discusses profiling PlayStation code and associated special cases.

The sections in this chapter are:

- Profiler Libraries and Interface Files (PlayStation)—the libraries and interface files that you add to your code in order to use the profiler
- Implementing the Profiler Library (PlayStation)—how to add the Profiler Library to your project and generate a profile
- Using the Profiler (PlayStation)—precautions to follow when analyzing profile data

Profiler Libraries and Interface Files (PlayStation)

The profiling code that actually keeps track of the time spent in a routine exists in two libraries: `PROFILE.C` and `PROFILE.H`. These libraries must be linked into your PlayStation application. A `PFD` file is generated when the two libraries are added to your code. You can then use the Profiler application to read and analyze the profile information contained in the `PFD` file.

Implementing the Profiler Library (PlayStation)

The profiler library is a body of code that records function calls and their duration. Follow the steps below to implement this functionality in your code:

1. **Open your project**

   Using the CodeWarrior IDE, open the project that you want to profile.

2. **Add `PROFILE.C` to your project.**

3. **Add `#include "Profile.h"` to your `PS_Debug_Prefix.h` file**

4. **Create a buffer in your application**

   In your application, create a buffer for saving profile data. It can be allocated at runtime, like this:

   ```
   ProfileBuffer = (char *)malloc(8192);
   ```

   The buffer can also be simply defined as part of the application:

   ```
   char ProfileBuffer[8192];
   ```

5. **Call `InstallProfiler()` in your code**

   Somewhere in your application's initialization routines, call `InstallProfiler()`:

   ```
   InstallProfiler(ProfileBuffer, SizeOfMem,
   TimerToUse, ExecutableName);
   ```

   `ProfileBuffer` is the buffer you created in step 4.

   `SizeOfMem` is the size of the buffer you created in step 4.

   `TimerToUse` is the timer you want the profile library to use.

NOTE: The profiler library can only use either `RCntCNT1` or `RCntCNT2`. Select the timer that your application is not using.

`ExecutableName` is the name of your executable, as entered in the MIPS Target settings panel, without the `.PSE` extension. For example, if `PS.PSE` were the name entered into the File Name edit field, then you would pass "`PS`" as the `ExecutableName`.

The example below successfully initializes the profiler library.

```
#define SIZE_OF_PROFILE_BUFFER       8192
#define NAME_OF_EXECUTABLE           "PS"
char ProfileBuffer[SIZE_OF_PROFILE_BUFFER];

InstallProfiler(ProfileBuffer, SIZE_OF_PROFILE_BUFFER, RCntCNT1,
NAME_OF_EXECUTABLE);
```

6. Add the recording calls from the profiler library to your code

NOTE: Starting with CodeWarrior for PlayStation 4, these calls will be automatically inserted into your code, making this step of the profiling process unnecessary.

The PlayStation Profiler Library records timing information at the start and end of each function being profiled. Consequently, you must add the `OnProcEntry()` and `OnProcExit()` calls to each function you wish to profile.

For example:

```
void foo(void)
{
  OnProcEntry();     //Do this before anything else.
.
.
.
  OnProcExit();     //Do this last, before returning.

  return;
}
```

7. }Add the Start- and Stop-Profiling calls from the profiler library to your code

To start profiling within your code, simply call

```
TurnOnProfiler();
```

You can do this at the start of your main game loop, or you may trigger it from a specific controller pad input:

```
If(ControllerInput & START_PROFILING)
TurnOnProfiler();
```

Once the profiler has started, it begins recording data and storing it into the buffer passed at the installation step (Step 5.).

After the profiler fills up the buffer, it writes out the profiled data to a file on your hard drive, within the same directory as your executable file. This file has the same name as your executable, only with a `.PFD` extension.

After the file has been written out, the profiler begins filling up its buffer once again. Every time the buffer is filled up, the profiler stops and records all its data to the end of its `PFD` file, then restarts again.

To turn off profiling within your code, simply call

```
TurnOffProfiler(PROFILER_WRITE_REPORT);
```

Again, you can do this at the end of your main game loop, or you may trigger it from a specific controller pad input:

```
If(ControllerInput & STOP_PROFILING)
TurnOffProfiler(PROFILER_DONT_WRITE_REPORT);
```

The argument passed in tells the profiler whether or not to write out the `PFD` file.

In addition, your code can call

```
Status = ProfilerStatus();
```

This will return PROFILING_ON or PROFILING_OFF, depending upon the state of the profiler. You can use this to aid in turning profiling on or off:

```
If((ControllerInput & START_PROFILING) &&
    ProfilerStatus() == PROFILING_OFF)
```

```
    TurnOnProfiler();
.
.
.
If((ControllerInput & STOP_PROFILING) &&
    ProfilerStatus() == PROFILING_ON)
    TurnOffProfiler(PROFILER_DONT_WRITE_REPORT);
```

8. **Build and run your application, ensuring that the Profiler Library code is called and that a `PFD` file is generated**

   After successfully generating a `PFD` file, you now need to analyze the data created by the Profiler Library. To do that, you must use MW Profiler.

Using the Profiler (PlayStation)

There are a few caveats to using the Profiler application correctly when profiling PlayStation code:

1. The `.PFD` and the `.PSE` files must have the same file name for the Profiler to properly display all pertinent information. Additionally, they must be in the same directory.

   To illustrate, if you load `EXAMPLE.PFD` into the Profiler application, it will look in the same directory for `EXAMPLE.PSE`. If the Profiler application cannot find the associated `PSE` file, it will not be able to display the names of the functions profiled. Instead, the address of the function calls will be displayed.

2. The Profiler application will only work properly if the `PSE` file generated the loaded `PFD` file. In other words, don't generate the `PFD` file, then rebuild your application, then use the Profiler application to analyze the `PFD` file. If you do this, the addresses of the functions will be different and the Profiler application will not be able to display the names of the functions properly. If you need to save a profile analysis, choose **Print** from the **File** menu to get a printed report. Alternatively, you can use the **Save** or **Save As** menu selections to save a text file of the analysis.

Total Time and the % columns

Both % columns indicate a percentage of the "Total Time". As calculated by the PlayStation Profiler Application, the Total Time is the total of all time spent by all function calls measured in the loaded `PFD` file. Notice that this is ***not*** the same as the total time your application spends running! This total time does not take into account the time spent within your application when the profiler was turned off, or was otherwise not recording data.

There are advantages to this approach, as it allows you to focus your profiling efforts on the small portions of your code that invariably are the most critical. For example, it's usually not very important to you how long the initialization of your game takes, or how much CPU time your menuing functions use. On the other hand, the performance of the rendering portion of your main game loop is probably very important to you, and you would like to be able to profile this portion of your code without watering down the analysis with other data.

See also "Viewing Results" on page 31.

7

Troubleshooting

This chapter contains frequently asked questions (and answers) about the Profiler. If you have a problem with the profiler, come here first. Others may have encountered similar difficulties, and there may be a simple solution.

The general topics that are covered include:

- Profile Times Vary Between Runs
- Problems while Profiling Inline Functions
- Profiling Library Could not be Found
- Profiler and Virtual Memory (Mac OS)
- Names are Garbled when Viewing a 68K Profile Dump (Mac OS)

Profile Times Vary Between Runs

Problem

I'm getting different results (within 10%) in MW Profiler every time I run my program.

Background

There are two potential reasons that this may be happening. Both are time-related problems. The first problem that can occur is inadequate time in the function relative to the profiler resolution. The second problem is clock resonance.

Inadequate time in the function

If the function time that you are trying to measure is only 10 times greater than the resolution of the timebase you are using, you'll have this problem.

Mac OS The 68K profiler uses these timebases:

Name	Resolution	Code
Ticks	~16700 µs	68K and PPC
Time Manager	~ 20 µs	68K and PPC
Microseconds	~ 20 µs	68K and PPC

Solution

To solve this problem, increase the number of times your function is called, then the average the profiler computes will be more accurate.

Sometimes it is helpful to pull a routine out of a program, and into a special test program which calls it many times in a loop for performance tuning purposes. However, this technique is susceptible to cache differences between the test and real program.

Clock resonance

If the operations you are performing in your profiled code coincide with the incrementing of the profiler clock, the results can be distorted, and could show wild variations.

Solution

One way to avoid this problem is to increase the number of times your function is called.

Mac OS Another way to avoid this problem is to increase the accuracy of the clock the profiler is using. You can only do this with the PowerPC profiler. The PowerPC profiler can use the `RTC` or `TB` registers of the PowerPC chip which have 128 ns or better resolution. Make a PowerPC version of your project and use `bestTimeBase` in your call to `ProfilerInit()`.

Problems while Profiling Inline Functions

Problem

My inline functions are not getting inlined when I'm profiling my code. What's happening?

Background

When the compiler switch for profiling is turned on, the default setting for "don't inline functions" is changed to true. This is so that these functions will have profiling information collected for them.

Solution

Place a `#pragma dont_inline off` in your source file to turn on function inlining again. You will not collect profile information for inline functions. In effect, a function can be inlined or profiled, but not both. The profiler cannot profile an inlined function.

TIP: If you use the `#pragma dont_inline off` in your code, you may see profile results for some inline functions.
When you declare an inline function, the compiler is allowed, but not required to inline the function. It is perfectly legal for the compiler to inline some functions, but not others. Data is collected only for the calls that were not inlined. The calls that were inlined have their time added into the time of the calling function.

Profiling Library Could not be Found

Problem

While trying to profile my dynamically linked library (shared library), I get an error message saying that the profiling library could not be found.

Background

This problem occurs when trying to use the profiling library to profile your dynamically linked library and the profiling library is not in the search path.

Solution

Add the profiling library to the search path. If you are using the CodeWarrior IDE, see the *CodeWarrior IDE User's Guide* for informa-

tion on search paths. If you are using MPW, see your *MPW User's Manual*.

Mac OS `ProfileLib` is a shared library and should be in your Extensions folder. Make sure a path to the Extensions folder is set for your project.

Profiler and Virtual Memory (Mac OS)

Profiling code with virtual memory or RAM Doubler active produces erratic and skewed result times. Make sure virtual memory or RAM Doubler are disabled when profiling your code.

TIP: It's actually best to turn off all extensions when performing a profile of your code. This will eliminate any chances of extension problems and background tasks skewing the profile results. Restart your computer with the Shift key held down to disable extensions.

Names are Garbled when Viewing a 68K Profile Dump (Mac OS)

Problem

I see garbage names in MW Profiler on 68K (non-CFM) profile runs.

Background

You've left calls to `UnloadSeg()` in your program. Calls to `UnloadSeg()` in a 68K application cause serious problems for the profiler. The profiler maintains pointers to locations in the code. If you unload a segment that has profiled routines, that code is free to move around in memory, or be purged. At that time the profiler's internal pointers become invalid. This is the leading cause of corrupted profile data files.

Solution

Remove, or comment out, the calls to `UnloadSeg()` when profiling your code.

8

Profiler Reference

This chapter contains the detailed technical reference information you may need when using the profiler.

The topics discussed include:

- Compiler Directives (Mac OS)—handling compiler directives in Mac OS code
- Memory Usage (Mac OS)—understanding memory usage in Mac OS code
- Time and Timebases (Mac OS)—the available time resolutions for Mac OS code
- Profiler Menu Reference—a reference for the main menus in MW Profiler
- Profiler Function Reference—a reference for all of the profiler API functions

Compiler Directives (Mac OS)

You can control routine-level profiling using compiler directives.

The C/C++ compiler supports three preprocessor directives that you can use to turn compiling on and off at will.

`#pragma profile on`	enables calls to the profiler in functions that are declared following the pragma
`#pragma profile off`	disables calls to the profiler in functions that are declared following the pragma
`#pragma profile reset`	sets the profile setting to the value selected in the preferences panel

You can use these directives to turn profiling on for any functions you want to profile, regardless of the settings in the Processor preferences. You can also turn off profiling for any function you don't want to profile.

The Pascal compiler implements two directives to control the compile-time generation of calls to the profiler.

`{$PROF +}`	enables calls to the profiler in procedures and routines that are declared following the directive
`{$PROF -}`	disables calls to the profiler in procedures and routines that are declared following the directive

The status of the profiler directive is saved and restored with the `{$PUSH}` and `{$POP}` directives.

Testing for the Profiler

As there are compiler directives to turn the profiler on and off, there are also directives to test if the profiler is on. You can use these tests in your code so that you can run your program with or without the profiler and not have to modify your code each time.

In C/C++, use the `#if-#endif` clause. For example:

```
void main()
{
#if __profile__    // is the profiler on?
  if (!ProfilerInit(collectDetailed, bestTimeBase, 20, 5))
  {
#endif
    test(15);
#if __profile__
    ProfilerDump("\pExample.prof");
    ProfilerTerm();
  }
#endif
}
```

In Pascal, use the following test:

```
{$IFC OPTION(PROF)}
  if (ProfilerInit(collectDetailed, bestTimebase,
             100, 20) = noErr) then begin
{$ENDC}
```

See also "Routine-level profiling" on page 26.

Memory Usage (Mac OS)

The profiler allocates two buffers in temporary memory to hold data as it collects information about your code: one based on the number of routines, and one based on the stack depth. You pass these parameters in your call to `ProfilerInit()`.

In summary collection mode, the profiler allocates 64 bytes * `numFunctions` and 40 bytes * `stackDepth`.

In detailed collection mode, the profiler allocates 12 * 64 * `numFunctions` bytes and 40 * `stackDepth` bytes.

As an example, assume `numFunctions` is set to 100, and `stackDepth` to 10. In summary mode the profiler allocates buffers of 6,400 bytes and 400 bytes. In detailed mode it allocates buffers of 76,800 bytes and 400 bytes.

`ProfilerGetDataSizes()` lets you query the profiler for the current size of the data collected in the function and stack tables. This information can be used to tune the parameters passed to `ProfilerInit()`.

See also "ProfilerInit()" on page 114.

Time and Timebases (Mac OS)

The CodeWarrior profiler supports four timebases. A timebase is the clock interval used to measure time in a routine. The shorter the interval, the more precise the measurements.

When you call `ProfilerInit()` you specify the desired timebase. The constant `bestTimeBase` tells the profiler to figure out the most precise timebase available on your platform and to use it.

Mac OS The following is a description of each timebase:

`bestTimeBase`	This is the preferred timebase. Using this timebase, the profiler selects the timebase with the highest resolution for your machine.
`PPCTimeBase`	This timebase is only available on a Power Macintosh from PowerPC code. It cannot be used in 68k code. It uses the built-in timing facilities of the PowerPC processor. On a 601 it uses the RTC registers. On subsequent PowerPC processors it uses the TB registers. This timebase is very low overhead and high accuracy. If this timebase is specified but not available, `ProfilerInit()` returns `paramErr`.
`microseconds-TimeBase`	This timebase uses the `_Microseconds` trap to measure time. It has the same accuracy as the `timeMgrTimeBase`, but less overhead (one trap vs. three traps). This timebase may not be available on all Macintosh computers. If this timebase is specified but not available, `ProfilerInit()` returns `paramErr`.
`timeMgrTime-Base`	This timebase uses the microseconds timing ability of the Time Manager. This is more accurate than the `ticksTimeBase`, (~20 μs), but has more overhead. On a Power Macintosh running System 7.1.2, the traps called run emulated, so there are three mixed mode switches for every routine call. This timebase is available on all Macintoshes with System 7.0 or greater.
`ticksTimeBase`	This counter increments 60 times a second. It is a very low overhead timebase with coarse accuracy. This timebase is available on all Mac OS computers.

Profiler Menu Reference

This reference section discusses the functionality of each menu item in MW Profiler, the data viewing application for the CodeWarrior profiler system.

These are the main menus:

- About MW Profiler—learn about MW Profiler
- File Menu—open, close, save, and print profiler data files, and quits the application
- Edit Menu (Mac OS)—provided for compatibility with other applications, no item is functional in MW Profiler
- View Menu—allows you to control the order and appearance of data in the MW Profiler window
- Windows Menu (Mac OS)—a list of all open profiler data files

About MW Profiler

Choosing **About Metrowerks Profiler** displays copyright and author information about the application.

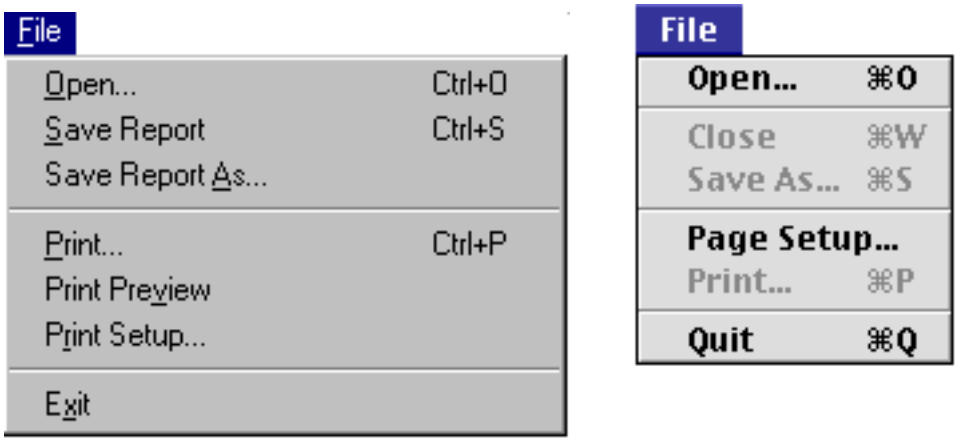
Windows The menu command **About Metrowerks Profiler** is in the **Help** menu.

Mac OS **The menu command About Metrowerks Profiler** is in the Apple menu.

File Menu

The **File** menu contains commands to open, close, save, and print documents, and to quit the application, as illustrated in Figure 8.1.

Figure 8.1 The File menu



Open displays the standard open file dialog that allows you to select and open an existing MW Profiler document.

Close closes the active window.

Save Report As or **Save As** saves the contents of the active window as a tab-delimited text document. The text file is organized the same way that the window is organized.

Page Setup or **Print Setup** displays the standard page setup dialog for the current printer.

TIP: To fit the profiler output across one page, select landscape orientation and 85% reduction.

Print Preview shows what the data file will look like when you print it out on paper.

Print prints the MW Profiler document in accord with the current view and sort order in the window.

Quit or **Exit** quits the MW Profiler application.

Edit Menu (Mac OS)

All **Edit** menu items are disabled. There is nothing to cut, copy, paste, clear, or select in an MW Profiler window.

View Menu

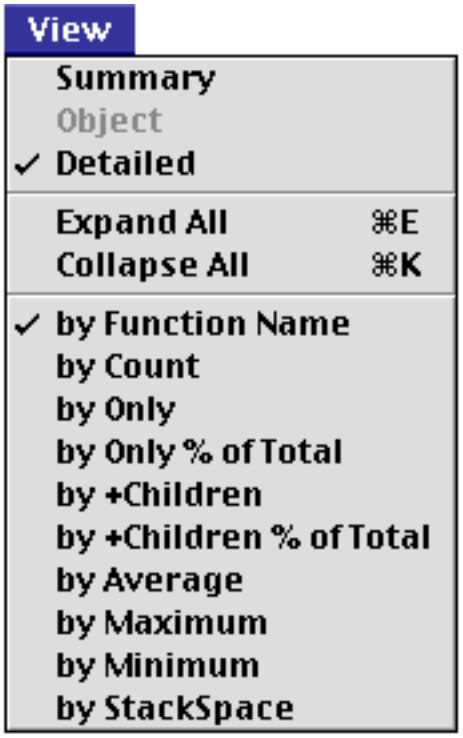
The **View** menu functions differently depending on the host you use to run MW Profiler.

Windows The **View** menu allows you to toggle the status bar on and off. The status bar is at the bottom of the MW Profiler window, and describes varioius features of the application.

Mac OS The **View** menu contains commands to sort and view MW Profiler data. The currently active option has a check mark in front of the menu item. You may also click on the column header to sort the data by the values in that column. Figure 8.2 shows the options that you can choose from.

- **Summary** displays the active profiler window in summary view. Summary view consists of one row for each routine in the profile data. Summary view is available for any profile.
- **Object** displays the active profiler window in object view. Object view is a summary of each class. Each class can be expanded to show the detail of each method within the class. Object view requires that the profile contain one or more mangled C++ class method names.
- **Detailed** displays the active profiler window in detailed view. Detailed view consists of one row for each path in the call tree. Detailed view requires that the `collectDetailed` method be used.
- **Expand All** expands all rows in an Object or Detailed view.
- **Collapse All** collapses all rows in an Object or Detailed view.
- **by Function Name** sorts the data in the active window by routine name.
- **by Count** sorts the data in the active window by count.
- **by Only** sorts the data in the active window by the Only time.

Figure 8.2 The View menu (Mac OS)



- **by Only % of Total** sorts the data in the active window sorted by Only percentage. This has the same effect as sorting by Only.
- **by +Children** sorts the data in the active window by +Children time.
- **by +Children % of Total** sorts the data in the active window by +Children percentage. This has the same effect as sorting by +Children.
- **by Average** sorts the data in the active window by Average time.
- **by Maximum** sorts the data in the active window by Maximum time.
- **by Minimum** sorts the data in the active window by Minimum time.
- **by Stack Space** sorts the data in the active window by Stack Space.

Windows Menu (Mac OS)

The windows menu lists all the open profiler data windows.

Profiler Function Reference

This is a reference for all Profiler functions mentioned in the text of this manual. The functions described in this chapter are:

- ProfilerInit()
- ProfilerTerm()
- ProfilerSetStatus()
- ProfilerGetStatus()
- ProfilerGetDataSizes()
- ProfilerDump()
- ProfilerClear()
- ProfilerCreateThread()
- ProfilerDeleteThread()
- ProfilerSwitchToThread()
- ProfilerGetMainThreadRef()
- InstallProfiler()
- UninstallProfiler()
- OnProcEntry()
- OnProcExit()
- TurnOnProfiler()
- TurnOffProfiler()
- ProfilerStatus()

Profiler API

The discussion of each function includes the following attributes:

- Description: A high-level description of the function

- Compatibility: Indicates the operataing system(s) and/or chip(s) with which the function is compatible.
  - Compatible targets are in black text
  - Incompatible targets appear in grey
  - Blank cells appear in the table in support of future targets
  - Win32 represents Windows95 and WindowsNT operating systems on x86 processors
  - Mac OS represents the Mac OS operating system on either PowerPC or 68K processors
  - PS OS represents the Sony PlayStation operating system
- Prototypes: The entire C/C++ and Pascal prototypes for the function
- Remarks: Implementational or other notes about the function

`ProfilerInit()`

Description

`ProfilerInit()` prepares the profiler for use and turns the profiler on. The parameters tell the profiler how this collection run is going to operate, and how much memory to allocate. `ProfilerInit()` *must* be the first profiler call before you can call any other routine in the profiler API.

Compatibility

This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes

C/C++:

```
typedef enum {
  collectDetailed,
  collectSummary
} ProfilerCollectionMethod;

typedef enum {
  ticksTimeBase,
  timeMgrTimeBase,
  microsecondsTimeBase,
  PPCTimeBase,
  bestTimeBase
```

```
} ProfilerTimeBase;

pascal OSErr ProfilerInit(
    ProfilerCollectionMethod method,
    ProfilerTimeBase timeBase,
    short numFunctions,short stackDepth);
```

Pascal:

```
const
{ for the mehod in PrifilerInit }
    collectDetailed = 0;
    collectSummary = 1;
{ for the timeBase in ProfilerInit }
    ticksTimeBase = 0;
    timeMgrTimeBase = 1;
    microsecondsTimeBase = 2;
    PPCTimeBase = 3;
    bestTimeBase = 4;

FUNCTION ProfilerInit(
    method: INTEGER;
    timeBase: INTEGER;
    numFunctions: INTEGER;
    stackDepth: INTEGER) : OSErr;
```

Remarks `ProfilerInit()` attempts to allocate memory in the Process Manager heap (temporary memory). If it can't get the required memory in that heap, it tries in the current heap. This strategy minimizes the effect that the profiler has on the application's memory partition.

`ProfilerInit()` can return memory manager errors such as `memFullErr` and `paramErr`. If `paramErr` is returned, it means that one of the constants passed in was out of range, or that `ProfilerInit()` has already been called.

The `method` and `timeBase` parameters select the appropriate profiler options. The `numFunctions` parameter indicates the number of routines in the program for which the profiler should allocate buffer storage. If the profiler is operating in detailed mode, this number is internally increased (exponentially), because of the branching factors involved. The `stackDepth` parameter indicates how many routines deep the stack can get.

A call to `ProfilerInit()` must be followed by a matching call to `ProfilerTerm()`.

`ProfilerInit()` is similar to `InstallProfiler()` for the PlayStation.

`ProfilerTerm()`

Description

`ProfilerTerm()` stops the profiler and deallocates the profiler's buffers. It calls `ProfilerDump()` to dump out any information that has not been dumped. `ProfilerTerm()` must be called at the end of a profile session.

Compatibility

This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes

C/C++:

```
void ProfilerTerm( void );
```

Pascal:

```
PROCEDURE ProfilerTerm;
```

Remarks

If a program exits after calling `ProfilerInit()` without calling `ProfilerTerm()`, timers may be left running that could crash the machine.

`ProfilerTerm()` is similar to `UninstallProfiler()` for the PlayStation.

`ProfilerSetStatus()`

Description

`ProfilerSetStatus()` lets you turn profiler recording on and off in the program. This makes it possible to profile specific sections of your code such as screen redraw or a calculation engine. The profiler output makes more sense if the profiler is turned on and off in the same routine, rather than in different routines.

Compatibility

This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes

C/C++:

```
pascal void ProfilerSetStatus( short on );
```

Pascal:

```
PROCEDURE ProfilerSetStatus( on: INTEGER );
```

Remarks

This routine and `ProfilerGetStatus()` are the only profiler routines that may be called at interrupt time.

Pass 1 to turn recording on and 0 to turn recording off.

`ProfilerSetStatus()` is similar to `TurnOnProfiler()` and `TurnOffProfiler()` for the PlayStation.

`ProfilerGetStatus()`

Description

`ProfilerGetStatus()` lets you query the profiler to determine if it is collecting profile information.

Compatibility

This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes

C/C++:

```
pascal short ProfilerGetStatus( void );
```

Pascal:

```
FUNCTION ProfilerGetStatus: INTEGER;
```

Remarks

This routine and `ProfilerSetStatus()` are the only profiler routines that may be called at interrupt time.

`ProfilerGetStatus()` returns a 1 if the profiler is currently recording, 0 if it is not.

`ProfilerGetStatus()` is similar to `ProfilerStatus()` for the PlayStation.

`ProfilerGetDataSizes()`

Description

`ProfilerGetDataSizes()` lets you query the profiler for the current size of the data collected in the function and stack tables. This

information can be used to tune the parameters passed to `ProfilerInit()`.

Compatibility This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes **C/C++:**

```
pascal void ProfilerGetDataSizes(
            long *functionSize,
            long *stackSize);
```

Pascal:

```
PROCEDURE ProfilerGetDataSizes(
    VAR functionSize: LONGINT;
    VAR stackSize: LONGINT);
```

Remarks If you have passed `collectDetailed` to `ProfilerInit()`, `ProfilerGetDataSizes()` returns the number of actual routines in the table, which may be larger than the value passed to `ProfilerInit()` in `numFunctions`. This is because the profiler multiplies `numFunctions` by 12 when it allocates the table. The multiplication is done so that you can easily switch between `collectDetailed` and `collectSummary` methods without changing the parameters.

`ProfilerDump()`

Description `ProfilerDump()` dumps the current profile information without clearing it. The filename passed must be a Pascal string.

Compatibility This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes **C/C++:**

```
pascal OSErr ProfilerDump( StringPtr filename );
```

Pascal:

```
FUNCTION ProfilerDump(filename: Str255) : OSErr;
```

Remarks This can be useful for dumping intermediate results on a long task. If the specified file already exists, a new file is created with an incrementing number appended to the filename. This allows the dump to be called inside a loop with a constant filename.

`ProfilerDump()` can return memory manager errors such as `memFullErr`, or file system errors.

`ProfilerClear()`

Description `ProfilerClear()` clears any profile information from the buffers.

Compatibility This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes

C/C++:

```
pascal void ProfilerClear( void );
```

Pascal:

```
PROCEDURE ProfilerClear;
```

Remarks `ProfilerClear()` retains the settings of `collectionMethod` and `timeBase` that were set by `ProfilerInit()`. It does not deallocate the buffers.

`ProfilerCreateThread()`

Description Use `ProfilerCreateThread()` to create a profiler thread structure to track the information in a new thread. You should call this routine whenever you create a new thread. This call creates the necessary profiler buffers and returns a reference used by the profiler. The `threadRef` provided by this routine must be stored with the thread and passed to `ProfilerSwitchToThread()` in the thread's swapIn proc.

Compatibility This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes **C/C++:**

```
typedef unsigned long ProfilerThreadRef;
pascal OSErr ProfilerCreateThread(
    short stackSize, long byteStackSize,
    ProfilerThreadRef *threadRef);
```

Pascal:

```
TYPE
  ProfilerThreadRef = LONGINT;
FUNCTION ProfilerCreateThread(
    stackSize: INTEGER; byteStackSize: LONGINT;
    VAR threadRef: ProfilerThreadRef) : OSErr;
```

Remarks The `stackSize` parameter is the number of profiler frames to use. The `byteStackSize` parameter is the size in bytes of the stack that was created. This is done so that you can use these routines with any threads package, the profiler does not include any explicit calls to the Thread Manager.

This routine allocates memory in the current heap.

`ProfilerDeleteThread()`

Description When a thread terminates, call `ProfilerDeleteThread()` to clean up the profiler thread tracking structures.

Compatibility This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes **C/C++:**

```
pascal void ProfilerDeleteThread(
                ProfilerThreadRef thread);
```

Pascal:

```
PROCEDURE ProfilerDeleteThread(
        thread : ProfilerThreadRef);
```

Remarks Returns `paramErr` if the `ProfilerThreadRef` passed in is not a `threadRef` returned from `ProfilerCreateThread()`.

`ProfilerSwitchToThread()`

Description Call this routine from the swapIn proc of the thread, passing the `threadRef` returned from `ProfilerCreateThread()`.

Compatibility This function is compatible with the following targets:

Win32	**Mac OS**	PS OS					

Prototypes **C/C++:**

```
pascal void ProfilerSwitchToThread(
                ProfilerThreadRef thread);
```

Pascal:

```
PROCEDURE ProfilerSwitchToThread(
        thread : ProfilerThreadRef);
```

Remarks Returns `paramErr` if not a `threadRef` returned from `ProfilerCreateThread()`.

`ProfilerGetMainThreadRef()`

Description Return the `threadRef` of the implicitly created main thread. `threadRef` is a pointer to an internal data structure.

Compatibility This function is compatible with the following targets:

Win32	**Mac OS**	PS OS					

Prototypes **C/C++:**

```
pascal ProfilerThreadRef
ProfilerGetMainThreadRef();
```

Pascal:

```
FUNCTION ProfilerGetMainThreadRef() :
ProfilerThreadRef;
```

Remarks Use this call to get the `threadRef`. Necessary to switch back to the main thread with `ProfilerSwitchToThread()`.

TIP: PowerPlant has been set up to make the profiler thread calls for you. Look at the threads classes for examples on how to use the profiler thread routines.

`InstallProfiler()`

Description This function initializes the profiler data and tells the profiler where in RAM to store the data and how big the data is. The `TimerToUse` is either `RCntCNT1` or `RCntCNT2`. If neither one is entered, `InstallProfiler()` returns `PROFILER_BAD_TIMER`. The `ExecutableFileName` is the name of the executable file (the default is "PS"). Look in the project settings, on the MIPS Project page to see what this is.

Compatibility This function is compatible with the following targets:

Win32	Mac OS	**PS OS**					

Prototypes **C/C++:**

```
int InstallProfiler(
  __declspec(bss) void *MemToUse,
  unsigned long SizeOfMem,
  unsigned long TimerToUse,
  char *ExecutableFileName);
```

Remarks See "Call InstallProfiler() in your code" on page 94 to see an example of how to use `InstallProfiler()` in your code.

If there was a problem with any of the arguments, `InstallProfiler()` returns `PROFILER_NOT_INITIALIZED`. If not, it returns `PROFILER_OK`.

`InstallProfiler()` is similar to `ProfilerInit()` for Mac OS.

`UninstallProfiler()`

Description This function turns the profiler off for good.

Compatibility This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes

C/C++:

```
void UninstallProfiler(void);
```

Remarks `UninstallProfiler()` assumes the caller will free up the RAM that was dedicated to it from `InstallProfiler()`.

`UninstallProfiler()` is similar to `ProfilerTerm()` for Mac OS.

`OnProcEntry()`

Description This function is called at the start of every function. It records profiling data and returns.

Compatibility This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes

C/C++:

```
extern void (*OnProcEntry)(void);
```

Remarks See "Add the recording calls from the profiler library to your code" on page 95 to see an example of how to use `OnProcEntry()` in your code.

OnProcExit() must be paired with `OnProcEntry()`. You must add both calls to each function you wish to profile.

`OnProcExit()`

Description This function is called at the end of every function. It records profiling data and returns.

Compatibility This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes **C/C++:**

```
extern void (*OnProcExit)(void);
```

Remarks See "Add the recording calls from the profiler library to your code" on page 95 to see an example of how to use `OnProcExit()` in your code.

OnProcEntry() must be paired with `OnProcExit()`. You must add both calls to each function you wish to profile.

`TurnOnProfiler()`

Description This function turns the profiler on.

Compatibility This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes **C/C++:**

```
int TurnOnProfiler(void)
```

Remarks See "}Add the Start- and Stop-Profiling calls from the profiler library to your code" on page 95 to see an example of how to use `TurnOnProfiler()` in your code.

If there is some problem, `TurnOnProfiler()` returns the appropriate error code. If there are no problems, then this function returns 0.

TurnOnProfiler() and TurnOffProfiler() are similar to ProfilerSetStatus() for Mac OS.

TurnOffProfiler()

Description This function turns the profiler off.

Compatibility This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes

C/C++:

```
int TurnOffProfiler(int WriteOutFile);
```

Remarks See "}Add the Start- and Stop-Profiling calls from the profiler library to your code" on page 95 to see an example of how to use TurnOffProfiler() in your code.

If the passed argument is PROFILER_WRITE_REPORT, this function writes out the profiler data file. Hence, TurnOffProfiler(PROFILER_WRITE_REPORT) causes the file to be generated, and TurnOffProfiler(PROFILER_DONT_WRITE_REPORT) does NOT write out the file. If there is some problem, TurnOffProfiler() returns the appropriate error code. If there are no problems, then the function returns 0.

TurnOffProfiler() and TurnOnProfiler() are similar to ProfilerSetStatus() for Mac OS.

ProfilerStatus()

Description This function returns the profiler status, either PROFILER_ON or PROFILER_OFF.

Compatibility This function is compatible with the following targets:

Win32	Mac OS	PS OS					

Prototypes

C/C++:

```
int ProfilerStatus(void);
```

Remarks See "}Add the Start- and Stop-Profiling calls from the profiler library to your code" on page 95 to see an example of how to use `ProfilerStatus()` in your code.

`ProfilerStatus()` is similar to `ProfilerGetStatus()` for Mac OS.

Index

Symbols

#pragma directives, profiler 105
__copy_vectors() 114, 122, 123, 124, 125
__PROFILE_ENTRY 59
__PROFILE_EXIT 59
__sinit 55

A

abnormal termination 57
accuracy, profiler 33
active profiler 17
API, including profiler 27
Apple menu 109
assembly language 84
asynchronous routines 57

B

bestTimeBase 49, 108
by +Children % of Total 112
by +Children command (Profiler) 112
by Average command (Profiler) 112
by Count command (Profiler) 111
by Function Name command (Profiler) 111
by Maximum command (Profiler) 112
by Minimum command (Profiler) 112
by Only % of Total command (Profiler) 112
by Only command (Profiler) 111
by Stack Space command (Profiler) 112

C

C/C++ API (Profiler) 27
Close command (Profiler) 110
code resources 55
CodeWarrior
    described 8
    installing ??–15
Collapse All command (Profiler) 39, 111
collection method 49
Compiler Directives 47
compiler directives 47, 48, 105

D

data
    exporting 36
    finding problems 40
    sorting 36, 111
    viewing 31
data columns
    contents 35
    resizing 34
debugging
    profiled code 58
design problems, finding 38
Detailed command (Profiler) 111
detailed data, collecting 49
detailed view 38
    finding design problems 38
directives
    C/C++ 105
    compiler 47, 48
    Pascal 106
display accuracy, profiler 33
dumping results 29

E

Edit menu 111
exceptions 57
Exit 110
exiting profiler 29
ExitToShell() 58
Expand All command (Profiler) 39, 111
exporting data 36

F

File menu 109
finding problems 40
function overflow 33
function-level profiling 26, 105

G

Generate Profiler Information 25, 71

I

initialize profiler 28

Inlining routines 79
installing CodeWarrior ??–15
interface files 27
interrupt tasks 57
interrupt time
    and profiler 117
interrupt time, and profiler 117

L

libraries 43, 44, 68, 93
longjmp() 57

M

MacApp profiling 56
Macintosh system requirements 14
main thread
    profiling 53
memory usage 49, 107
microsecond 34
microsecondsTimeBase 49, 108
millisecond 34

O

Object command (Profiler) 111
object performance 39
object view 39
Open command (Profiler) 31, 110
overflow functions 33
overflow stack 33

P

Page Setup command (Profiler) 110
Pascal API (Profiler) 28
passive profiler 16
PPCTimeBase 49, 108
preprocessor directives 47, 48
    C/C++ 105
    Pascal 106
Print command (Profiler) 110
Print Preview 110
Print Setup 110
printing profiler output 110
profiler
    accuracy 33
    active 17
    C/C++ API 27
    components 22
    defined 16
    early 16
    exiting 29
    getting results 29
    including API 27
    initialize 28
    libraries 44, 68
    memory usage 49
    Pascal API 28
    passive 16
    printing output 110
    recursive calls 46
    sampling 16
Profiler Function Reference 113
Profiler Init() 49
Profiler Tutorial (Mac OS) 60
Profiler window 32
    data columns 33–36
    header items 32
ProfilerClear() 119
ProfilerCreateThread() 52, 119
ProfilerDeleteThread() 53, 120
ProfilerDump() 118
ProfilerGetDataSizes() 117
ProfilerGetMainThreadRef() 53, 121
ProfilerGetStatus() 117
ProfilerInit() 114, 122, 123, 124, 125
    data buffer 33
    warning 58, 59
ProfilerSetStatus() 116
ProfilerSwitchToThread() 52, 121
ProfilerTerm() 116
    warning 58, 59
ProfilerThreadRef defined 120
profiling
    activating 25
    asynchronous routines 57
    by function 26, 105
    code resources 55
    exceptions 57
    far code 44
    inline functions 100

interface files 45
interrupt tasks 57
MacApp 56
main thread 53
near code 44
setjmp 57
shared libraries 54
smart code 44
static constructors 55
threads 52
Profiling C++ Static Constructors 55
Profiling MacApp 56
Project Settings 25

Q

Quit command (Profiler) 110

R

requirements *See* system requirements
results
dumping 29
exporting 36
finding problems 40
opening 31
sorting 36, 111

S

sampling profiler 16
Save As 110
Save As command (Profiler) 29, 36
Save Report As 110
saving results 29
setjmp() 57
shared libraries 54
sorting data 36, 111
stack overflow 33
stack space, finding problems 41
static constructors 55
StProfileSection 46
strength reduction 81
Summary command (Profiler) 111
summary data 49
summary view 37
system requirements 13–14, 14–??
Macintosh 14
Windows 13

T

Threads
Profiling 52
Viewing Results 53
tick, defined 34
ticksTimeBase 34, 49, 108
time hogs, finding 40
timebase 49, 107
defined 107
timeMgrTimeBase 49, 108

U

UnloadSeg() 60
UnloadSeg(), problems with profiler 52, 56, 60, 102

V

View 111
view in profiler
detailed 38
object 39
summary 37

W

what's in this manual 9
where to learn more 10
Windows system requirements 13

Index

CodeWarrior

Profiler Manual

Credits

writing lead: Derek Saldana

other writers: Jim Trudeau, Carl B. Constantine, John Roseborough, Roger Wong

engineering: Steve Nicolai

frontline warriors: Marc Paquette, Eric Scouten, Neal Trautman, Stephen Espy, Andrea Pellizzon, Terry Constantine, and CodeWarrior users everywhere

Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C/C++ front-end compiler
Pascal Compilers Reference	Information on the Pascal front-end compiler
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler syntax
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks ANSI standard C library
MSL C++ Reference	Function reference for the Metrowerks ANSI standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
The PowerPlant Book	Introductory guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting BeOS	How to use CodeWarrior to program for BeOS
Targeting Java VM	How to use CodeWarrior to program for the Java Virtual Machine
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting NEC V810/830	How to use CodeWarrior to program for NEC V810/830 processors
Targeting Net Yaroze	How to use CodeWarrior to program for Net Yaroze game console
Targeting Nucleus	How to use CodeWarrior to program for the Nucleus RTOS
Targeting OS-9	How to use CodeWarrior to program for the OS-9 RTOS
Targeting Palm OS	How to use CodeWarrior to program for PalmOS
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting VxWorks	How to use CodeWarrior to program for the VxWorks RTOS
Targeting Win32	How to use CodeWarrior to program for Windows